THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

# Agile cryptography

A universally composable approach

**Citation for published version:**
Badertscher, C, Ciampi, M & Kiayias, A 2023, Agile cryptography: A universally composable approach. in G Rothblum & H Wee (eds), *Theory of Cryptography: 21st International Conference.* 1 edn, vol. 14372, Lecture Notes in Computer Science, vol. 14372, Springer, pp. 480-509, 21st International conference on Theory of Cryptography Conference, Taipei, Taiwan, Province of China, 29/11/23. https://doi.org/10.1007/978-3-031-48624-1_18

**Digital Object Identifier (DOI):**
10.1007/978-3-031-48624-1_18

**Link:**
Link to publication record in Edinburgh Research Explorer

**Document Version:**
Early version, also known as pre-print

**Published In:**
Theory of Cryptography

**General rights**
Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

**Take down policy**
The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

OPEN ACCESS

Download date: 04. Jul. 2024

# Agile Cryptography: A Universally Composable Approach

Christian Badertscher[1], Michele Ciampi[2], and Aggelos Kiayias[3]

[1] *Input Output, Switzerland* – christian.badertscher@iohk.io
[2] *The Universtiy of Edinburgh, UK* – michele.ciampi@ed.ac.uk
[3] *The University of Edinburgh & Input Output, UK* – aggelos.kiayias@ed.ac.uk

**Abstract**

Being capable of updating cryptographic algorithms is an inevitable and essential practice in cryptographic engineering. This *cryptographic agility*, as it has been called, is a fundamental desideratum for long term cryptographic system security that still poses significant challenges from a modeling perspective. For instance, current formulations of agility fail to express the fundamental security that is expected to stem from timely implementation updates, namely the fact that the system retains some of its security properties provided that the update is performed prior to the deprecated implementation becoming exploited.

In this work we put forth a novel framework for expressing updateability in the context of cryptographic primitives within the universal composition model. Our updatable ideal functionality framework provides a general template for expressing the security we expect from cryptographic agility capturing in a fine grained manner all the properties that can be retained across implementation updates. We exemplify our framework over two basic cryptographic primitives, digital signatures and non-interactive zero-knowledge (NIZK), where we demonstrate how to achieve updateability with consistency and backwards-compatibility across updates in a composable manner. We also illustrate how our notion is a continuation of a much broader scope of the concept of agility introduced by Acar, Belenkiy, Bellare, and Cash in Eurocrypt 2010 in the context of symmetric cryptographic primitives.

## 1 Introduction

A lesson we all know well is that cryptographic implementations are not forever. And while in theory, switching between implementations may be as easy as wiping a whiteboard, in cryptographic engineering, transitioning between implementations of the same primitive can be very challenging. This was exemplified in the efforts of deprecating MD5 and SHA-1 that were brought about by the attacks of [WY05] and [WYY05] which eventually culminated to a complete collapse of collision resistance for these functions, [SSA+09] [SBK+17]. Despite significant communication efforts and the practical attacks that were developed, the functions lingered for years, see for example [Ras17].

Developing computer systems in a way that they are capable of updating the underlying cryptographic implementations they use has been an important engineering objective for over a decade. It is the main objective behind the concept of cryptographic *agility* [Sul09], that promotes software engineering practices that facilitate the easy swapping of cryptographic algorithms. Agility has been identified as a key requirement in standardization of cryptographic systems, see e.g., the NIST report on post-quantum security [CJL+16] and remains a much discussed topic in cryptographic engineering circles and one that is also, still, not sufficiently researched, cf. [OPM22].

Given the above state of affairs, an important question is whether there is a way to formally study implementation updates in cryptographic systems and even realize them in a safe manner. In [ABBC10] cryptographic agility is given a formal treatment identifying it as a special case of the problem of key-reuse between cryptographic algorithms. While key re-use is important security consideration in cryptographic systems and indeed highly relevant in the context of updates, we argue that cryptographic implementation updateability and agility itself, is a much broader and not sufficiently understood topic. The reason is that "agility" as defined in [ABBC10] *"is about individually secure schemes sharing a key. It is not about what happens when a scheme is broken and replaced by another that is (hopefully) secure."* However it is clear that in a broader cryptographic engineering context, the agility desideratum is exactly about broken cryptographic algorithms and the need to update them. Indeed, it is in this context that the term is mentioned in [Sul09] and [CJL+16], where the objective is to move from broken hash function implementations and, respectively, broken classical cryptographic algorithms, to ones that are more secure. As stated explicitly in [Sul09], agility is about *"an administrator [who] might choose to replace an algorithm that was recently broken with one still considered secure."*

At first sight, this direction might seem hopeless: a security collapse of a cryptographic algorithm seems catastrophic for security. Key material may be exposed, privacy of past transactions can be compromised and integrity of future interactions can be at risk. Furthermore, even attempting to model security in this setting seems problematic. How to model an event like an "efficient algorithm for factoring is just discovered" in the timeline of events of a cryptographic system? The cryptographic assumptions that are used to assemble the conditional statements in the security theorems we are so fond of have a truth-value that is universal and definite — it is not meant to change in the course of the system deployment based on our understanding.

Contrary to these theoretical barriers, security practitioners have no issue to envision the concept of a secure update. They know that taking advantage of the specifics of the system at hand, an update process can be followed to sanitize the parts that are deemed vulnerable and provided that such process is completed on time —prior to an algorithm compromise or other adversarial exploit— the system will retain a fair amount of its security properties. This final point is the main motivation behind our work that bridges the cryptographic theoretical design and modeling toolset with the engineering practices of software updates. We study the notion of cryptographic agility in UC which allows us to master the complexity of interactions present in computer systems where multiple cryptographic modules interact with each other. Such a composable definition of what a secure update system for a cryptographic task ideally is, allows us not only to theoretically reason about the security properties of update protocols, but also to rely on the composition theorem to construct complex systems with clear security semantics that contain updatable modules.

## 1.1 Contributions

We present a novel modeling of cryptographic agility within the setting of Universal Composition and illustrate it with concrete protocols for important cryptographic primitives. Our results include:

**A generic framework for updates.** Our model framework captures the concept of securely updating a cryptographic implementation in the form of a generic update functionality we denote by $\mathsf{U}^{\mathscr{F}}_{\mathtt{SttUp},\mathtt{UpPred}}$ that is parameterized by a class of ideal functionalities $\mathscr{F}$ as well as two programs `UpPred` and `SttUp`. Each member of the class $\mathscr{F}$ is indexed by a particular implementation and provides the same interface to the users who engage with it. At any given time the session participants of the functionality are connected to a particular member of $\mathscr{F}$ which initially is a designated member $\mathcal{F}_0$ for all parties. As time passes, participants, even in different sessions, may initiate an

update to another implementation, by pointing to another member of $\mathscr{F}$. While the interface of $\mathsf{U}^{\mathscr{F}}_{\mathtt{SttUp},\mathtt{UpPred}}$ contains that of the members of $\mathscr{F}$, the additional update interface of $\mathsf{U}^{\mathscr{F}}_{\mathtt{SttUp},\mathtt{UpPred}}$ enables users of the functionality to coordinate an update. The security properties of the update are captured by a *state-update* function `SttUp` and the *update* predicate `UpPred`. `SttUp` represents the (ideal) initialization of the state of the updated functionality (as a function of its history), and `UpPred` dictates the circumstances under which a party successfully completes the update:

- *Security across updates.* In the most simple parameterization, $\mathtt{UpPred} \equiv 1$ and $\mathtt{SttUp} \equiv \emptyset$, amounting to a "clean install", we have that the new primitive substitutes the old one without retaining any properties or guarantees from the past instantiation. There are many reasons that a "clean install" is a very limiting way of approaching the concept of updatable cryptographic primitives. One thing is that key material has to be re-generated. A more crucial downside however is that backwards compatibility is lost — something highly desirable from an engineering perspective and difficult to attain as pointed out in [JPS13]. This can be a very serious consideration since the updated system may still need to interoperate with components using the previous algorithm. Our framework captures naturally such considerations by suitably defining via the `SttUp` program which portion of the past state, such as of functionality $\mathcal{F}_{i-1}$, should be retained when updating to the functionality $\mathcal{F}_i$ during the $i$-th update.

- *Update coordination patterns.* Updating the implementation of a particular primitive requires some action by all participants engaged in a certain execution session. This may be done by varying degrees of coordination and our framework is capable of doing this by programming the `UpPred` predicate to grant the update successfully at the right conditions. For instance, the update predicate could demand that all parties transition in tandem, or could ensure the update unifies a set of parties currently working with different versions.

**Fine-grain corruption interface.** By specializing the UC corruption model to capture explicit subroutine corruptions, we are able to propose a unified modeling tool capturing the seemingly different security breaches of (1) corrupted implementations, (2) key leakage, or (3) failed security assumptions, without having to declare parties affected by such failures as *fully corrupted*. While such a fine-grain look at corruption is of independent interest, in the context of updates, such a formalization is important because it allows us to express advanced features of updates. For example, we can capture that if an update happens prior to any of the above failures, the system is uninterruptedly secure. We can also capture that as long as a party is not fully corrupted (but suffering from any of the above failures), an update regains the party's full security guarantees.

**Updatable Signatures and NIZKs.** We present two extensive studies applying our framework to the setting of digital signatures and non-interactive zero-knowledge schemes. In both cases, we explore the ramifications of the requirement of consistency, backwards compatibility across updates, and healing from sub-corruptions. In particular, the desideratum is that past signatures and NIZKs still verify after an update, while mitigating any side-effects by a collapsed cryptographic implementation, in the sense of regaining security after the update takes place. We achieve this via an update process that transfers information about the state onwards, while eliminating algorithmic dependencies on previous versions which may be compromised. In the signature case, we obtain a general and provably secure blueprint of how computer systems can become post-quantum ready today without sacrificing efficiency. In the NIZK case, we show that more effort is required by the

prover to ensure the preservation of soundness of the proof system across updates while ensuing that no leakage occurs due to the update system itself.

**Comparison with previous notion.** Finally, for completeness, we demonstrate how our framework subsumes the concept of agility as defined in [ABBC10]. Following that work, we focus on pseudorandom functions (PRFs) and we show what exact type of an updatable PRF functionality in our framework corresponds to their agility notion. Expectedly, the result does not account for (adaptive) corruptions since agility in [ABBC10] is equated to key reuse and hence it fundamentally requires the protection of the key as algorithms are being swapped across updates.

## 1.2 Related Work

As mentioned above, our work recasts the issue of cryptographic agility in the universal composition setting taking a broader view of what cryptographic agility means compared to [ABBC10] who consider it a special case of key sharing across primitives. This notion is subsumed within our framework, which in a similar manner, can also accommodate protocol substitution attacks [BPR14, FM18]. Our work also relates to key-insulated cryptosystems [DKXY03] and key-evolving cryptography [Fra06] in the sense that our notion can encompass forward security with previous key compromise. The important distinction though of our cryptographic agility framework is that we do not outsource the version coordination to the users and the fact that implementations of the underlying primitive can be independent; in contrast, key-insulated and key-evolving cryptographic primitives require a single monolithic implementation that has the users themselves specify at each invocation which version of the key they engage with. The case of "updatable encryption" [LT18, BDGJ20] is similar. Another treatment of a specific primitive in the context of software updates is given in the context of blockchain protocols in [CKKZ20]. In the context of key exchange [BBF$^+$16], agility has been casted as having a configurable selection of multiple protocol and cipher modes that can be chosen via running a negotiation protocol. In our agility framework such negotiations can take arbitrary form between parties and culminate to a particular input to the updatable functionality which will be then be responsible for facilitating the coordination steps needed for parties to implement the switch to the negotiated mode of operation. It follows that it is straightforward to express the concept of downgrade attacks as well as the concept of downgrade-resilience (cf. [BBF$^+$16]), by suitably restricting to environments that prohibit honest parties from "shooting themselves in the foot" and agreeing to switch (downgrade) to a weak implementation. Finally, we note that our topic bears a superficial resemblance to the concept of "cryptography with updates" [ACJ17] which in fact relates to incremental cryptography [BGG94], as well as to firewall-based constructions [CMNV22] that tame dishonest behavior by sanitizing the messages from a cryptographic protocol. Recently Poettering et al. [PR22] propose a forward secure signature scheme suitable to authenticate software versions by the vendors with two main features: first, assuming secure erasures, the scheme is forward secure, and second, based on a natural incentive structure, the scheme enjoys a particular self-enforcement mechanism that strongly disincentivizes coercion attacks in which vendors are forced, e.g. by nation state actors, to misuse or disclose their keys.

## 1.3 Notation

We denote the security parameter with $\lambda \in \mathbb{N}$. A randomized algorithm $\mathcal{A}$ is running in *probabilistic polynomial time* (PPT) if there exists a polynomial $p(\cdot)$ such that for every input $x$ the running time of $\mathcal{A}(x)$ is bounded by $p(|x|)$. We call a function $\mathrm{negl} : \mathbb{N} \to \mathbb{R}^+$ *negligible* if for every positive polynomial $p(\lambda)$ a $\lambda_0 \in \mathbb{N}$ exists, such that for all $\lambda > \lambda_0 : \epsilon(\lambda) < 1/p(\lambda)$. We denote by $[n]$ the set

$\{1, \ldots, n\}$ for $n \in \mathbb{N}$. We use "=" to check equality of two different elements (i.e. $a = b$ then...) and "$\leftarrow$" as the assigning operator (e.g. to assign to $a$ the value of $b$ we write $a \leftarrow b$). A randomized assignment is denoted with $a \xleftarrow{\$} \mathsf{alg}$, where $\mathsf{alg}$ is a randomized algorithm and the randomness used by $\mathsf{alg}$ is not explicit. If the randomness is explicit we write $a := \mathsf{alg}(x; r)$ where $x$ is the input and $r$ is the randomness. Let $\mathbf{v}$ be a sequence of elements (vector); by $\mathbf{v}[i]$ we mean the $i$-th element of $\mathbf{v}$. Analogously, for a bi-dimensional vector $M$, we denote with $M[i, j]$ the element identified by the $i$-th row and the $j$-th column of $M$.

# 2 UC Basics and Corruption Models

## 2.1 Overview

We use the UC framework [Can20] in this work and give here a brief overview.

*Protocol and protocol instances.* A protocol $\pi$ is an algorithm for a distributed system formalized as an interactive Turing machine (ITM) with several tapes. For this paper, we are mainly referring to three tapes: (1) the input tape, holding inputs written by a calling program, (2) the subroutine-output tape, which holds return values from called programs, and (3) the backdoor tape which formalizes the interaction of an ITM with the adversary (for example to capture corruption modes). Of great formal interest are the so-called *structured protocols* that are protocols that consist of a *shell* part that takes care of model-related instructions such as corruption handling, and a *body* part that encodes the actual cryptographic protocol. The body can again consist of a protocol (consisting of shell and body) which yields a sequence of shells. Towards defining a UC execution, UC defines an ITM instance (denoted ITI), which is defined as the pair $M = (\mu, \mathrm{id})$, where $\mu$ is the description of an ITM and $\mathrm{id} = (\mathsf{sid}||\mathsf{pid})$ is its identity, consisting of a session identifier $\mathsf{sid}$ and a party identifier $\mathsf{pid}$. An instance is associated with a Turing machine configuration, which can be seen as this machine's state. An *instance or session of a protocol* $\pi$, with respect to a session identifier $\mathsf{sid}$, is defined as a set of ITIs $(\pi, \mathrm{id}_1), (\pi, \mathrm{id}_2), \ldots$ with $\mathrm{id}_i = \mathrm{sid}||\mathrm{pid}_i$. Each such ITI is referred to as a (main) *party*, and the *extended instance* is the transitive closure of machines spawned as a consequence of running $\pi$, in particular, it encompasses all the subroutines.

*Ideal Functionalities.* A special type of protocols are ideal protocols that formalize the idealization of a cryptographic task. They are represented by an ideal functionality and its specification is usually referred to by $\mathcal{F}$. An instance of this functionality can be thought of as a "trusted third party". Parties are formally represented as dummy machines that forward inputs and outputs to and from this particular ITI, respectively. $\mathcal{F}$ therefore has to specify all outputs generated for each party, and the amount of information the ideal-world adversary learns (via the backdoor tape) and what its active influence is via its interaction with $\mathcal{F}$. Functionalities directly handle the corruption requests by an adversary (and usually adjust their behavior based on this information). We denote the corruption set maintained by parties by $\mathcal{C}$ and this will play an essential role in our treatment.

*Execution of a protocol, adversary, and corruption models.* In a UC execution, an environment is allowed to spawn a session of a protocol $\mu$. Often, this is either a real protocol $\pi$ or an instance of some ideal functionality running with dummy parties. Additionally, the environment is allowed to invoke the adversary. The adversary can communicate with other ITIs by writing (only) on their respective backdoor tapes. This tape is used to model security properties provided by functionalities (e.g., a secure channel could leak the length of the message via the backdoor tape). The backdoor tapes are also used to model party corruption. The corruption model is formally specified by how machines react to these messages on the backdoor tape. The plain UC model does not prescribe any

specific corruption model, but there is a standard corruption mode that we describe in more detail below in Section 2.2 as it is relevant to this work because we are going to propose an extension to it.

*UC emulation and composition.* While UC emulation is a very general notion, we are only interested in the special case that a protocol $\pi$ UC-realizes an ideal functionality $\mathcal{F}$: A protocol $\pi$ UC-realizes $\mathcal{F}$, if for any (efficient) real-world adversary $\mathcal{A}$ there is an (efficient) ideal-world adversary (the simulator) $\mathcal{S}$ such that no (efficient) environment can distinguish the (real) execution of protocol $\pi$ from the (ideal) execution of $\mathcal{F}$. This emulation notion is composable: if a protocol $\pi_1$ UC-realizes $\mathcal{F}_1$, and another protocol $\pi_2$ uses $\mathcal{F}_1$ as a subroutine to UC-realize some other functionality $\mathcal{F}_2$, then one can replace invocations of $\mathcal{F}_1$ by invocations to $\pi_1$ and the composed protocol (consisting of $\pi_2$ calling $\pi_1$) UC-realizes $\mathcal{F}_2$.

**Standard functionalities.** In this work, we will make use of the standard formalism of signatures, non-interactive zero-knowledge (NIZK), and pseudo-random functions (PRFs). We defer the preliminaries to Section A in the supplementary material.

## 2.2 Fine-grained Corruption Model

The standard way of modeling byzantine corruptions in UC has two important aspects: when a machine is corrupted, the adversary takes full control over the future input-output behavior(and learns all previous inputs and outputs). Second, the machine reports itself as corrupted to a *corruption aggregation ITI* that aggregates all corruption information of an extended session. In more detail, this aggregation machine records corruptions within the main session and is further made aware of all subroutines and their respective session-IDs in order to be able to (recursively) retrieve the corruption status via the corruption aggregation machines of those invoked (sub)sessions. In this way, the environment receives "genuine" information about the corruption set in an execution. The corruption aggregation is identified by a special PID $\mathscr{A}$. When invoked to report the corruption set, in the standard party-wise (pid-wise) corruption model, the machine reports the list of known party identities that are listed as corrupted.

This is however only a restricted use of the entire formalism put forth in [Can20]. In general, if a machine with (extended) identity $id := (\pi, \mathrm{sid}, \mathrm{pid})$ is registered as corrupted, the corruption-aggregation machine can store any function $f(id)$ to model a fine-grained level of corruption and not just report pid. Similarly, an ideal functionality, which models all interaction including the faithful reporting of the corruption information can report a more fine-grained set of corrupted machines.

In our model, we follow pid-wise corruption with the following simple addition. Let $\pi$ be a protocol and let $\mathcal{H}$ be a dedicated hybrid functionality invoked only by a single party (imagine $\mathcal{H}$ to be a private memory functionality of party pid). Then we allow the corruption aggregation machine to report two kinds of information: $f(\rho, s, p) = p$ if $\rho$ is any other code than $\mathcal{H}$ (as usual, the entire party is considered corrupted in this case). If $\rho$ corresponds to the ideal protocol for $\mathcal{H}$, then the entire identity is revealed, i.e., the session, the party-id and the label of the functionality. A protocol in this model has hence two corruption modes: session sid of $\pi$ run by party pid follows the standard corruption mechanism except that when its hybrid functionality $\mathcal{H}$ obtains the corruption message to corrupt party pid, the protocol reports $(\mathcal{H}, \mathrm{pid}, s)$ where $s$ is the session-id within which $\mathcal{H}$ was invoked. If $\mathcal{H}$ models a local memory, then this corruption mode allows to corrupt the memory of a party without corrupting the entire party.

**Ideal-world correspondence.** In the ideal world, the realized functionality $\mathcal{F}$ must support more complex corruption instructions to reflect the different corruption states that are possible based on the fine-grained information contained in $\mathcal{C}$ (note that since corruption is modeled in UC

as part of the protocol execution, it is the functionality that is responsible for the reporting in the ideal execution). Instead of just reacting on ordinary party corruptions, our ideal functionalities will incorporate specific instructions to reflect the more fine-grained model. For example, an ideal functionality $\mathcal{F}$ will allow backdoor messages from the adversary to specify that e.g. a memory of some party pid is corrupted, upon which the security would downgrade to the extent defined by the functionality $\mathcal{F}$, which is presumably less severe than when the functionality receives an ordinary request to corrupt the *full* party pid.

Recall from [Can20] that standard PID-wise corruption in the ideal world refers to a minimally supported interface of a functionality when a party $P$ is (fully) corrupted. In particular, at the point the party is corrupted via a corruption message on the functionality's backdoor tape, the functionality outputs to the adversary all the values received from $P$ and output to $P$ so far. External inputs to $P$ are ignored and given to the adversary, and the adversary is free to specify any message to be output by $P$ (that is, externally written by the dummy protocol for $P$ without changing the state of the functionality). Moreover, the adversary can decide to formally invoke the functionality with $(\textsc{input}, P, v)$ upon which the functionality executes its defined instructions for input $v$ as coming from $P$ (based on the current state which includes the corruptions status of parties).

### 2.3 Further Conventions Relevant to this Work

**State, behavior, and functionality classes.** Without loss of generality we assume that each instance of a functionality manages a *state* data-structure which encodes all the information relevant for its input-output behavior (such as inputs that the functionality receives and the output that it provides). We describe the state of a functionality by means of a variable $\mathsf{state}$. Note that the behavior of a functionality (defined as a Turing machine in UC) can equivalently be described using a state-transition model, i.e., as a sequence of conditional probability distributions (defined for output space $Y$, input space $X$ and state space $S$) $p^{\mathcal{F}}_{(Y,S_i)|(X,S_{i-1})}((y,\mathsf{state}_i)|(x,\mathsf{state}_{i-1}))$ for $i > 0$ assuming an initial well-defined state $\mathsf{state}_0$. The formal inputs correspond to the content of the respective input tapes and the outputs define the content on the outgoing message tape. We give a quick overview of this correspondence in Section B for completeness.

We introduce the notion of a *class of functionalities* $\mathscr{F}$, which in its full generality can be thought as being defined by a language $L \subseteq \{0,1\}^\star$ such that a functionality $\mathcal{F}$ belongs to $\mathscr{F}$ if the state $\mathsf{state}$ of any instance of $\mathcal{F}$ satisfies $\mathsf{state} \in L$ and its initial state (representing the initial configuration before any interaction) as $\perp$. Further, we assume that the corruption set (which the functionality has to export e.g. to the environment) is represented by an explicit state variable $\mathcal{C}$. That is, we write $\mathsf{state} = (\mathsf{state}', \mathcal{C})$ (initially, the corruption set is empty). We assume that we have a fixed party universe $\mathcal{P}$ and our functionalities interact with this set (of party identifiers). However, it is easy to extend the treatment to dynamic party sets [BCH$^+$20].

**Functionalities running arbitrary code.** We will specify functionalities that receive code or algorithm, say $\mathsf{alg}$, as part of their input from some caller which they are expected to execute on certain inputs $x$. In order for this to be unproblematic [Can20] defines how an ideal functionality evaluates $\mathsf{alg}(x)$: instead of running it internally, the functionality invokes a new subroutine (a new machine) that executes $\mathsf{alg}$. Since the runtime of any new machine depends on the so-called *import* it receives (until it runs out of import), the execution of $\mathsf{alg}$ can therefore be made safe as follows: whenever the party receives the request to execute $\mathsf{alg}$ on input $x$, it expects the calling party to provide the import for this, which it relays to the subroutine executing $\mathsf{alg}$. This way, the

algorithm's runtime uses up the runtime budget of the entity instructing the execution of $\mathsf{alg}$ on input $x$. In order to keep the descriptions simple, we assume the above mechanism implicitly.

# 3 Our Model for Updatable Functionalities

## 3.1 A Fine-Grained Corruption Model for Update Systems

Intuitively, we want to be able to capture sub-routine corruption (and not consider it as a full corruption) because it is updates that will make parties recover from sub-routine corruptions. To make this idea more precise, we introduce a generic modeling tool that captures subversion of any kind building on top of Section 2.2. That is, we introduce a machinery by which we formally model that a certain algorithm used by a party $P$ becomes insecure (without $P$ being corrupted). Our machinery can be seen as a particular way to model the leakage of secret keys, subverted and broken algorithms.

**Corruptions in the real world.** To model real-world attacks on specific cryptographic schemes (e.g., the adversary gets access to secret keys of the honest parties, or installs a malware into the machine of the honest users that changes the behavior of the cryptographic algorithms, or obtains access to inputs due to a broken algorithm), we introduce a new modeling technique.

We present a functionality $\mathcal{F}_{\mathsf{Comp}}$ (Figure 1) that internally runs arbitrary procedures and stores all the input-output pairs. This device is corruptible in the sense that upon corruption (with respect to a certain party $P$) the adversary gets full access to the memory and can also fully control it. It is a sub-routine corruption, and thus the party $P$ does not become corrupted automatically.

The functionality $\mathcal{F}_{\mathsf{Comp}}$ can be used as a modeling tool to express, in a fine-grained way, which parts of a system we do consider as potentially risky and vulnerable (it need not stand for a real-world object such as a computing module). This includes three important special cases:

- If we believe some cryptographic key material is more exposed than others (e.g., because a long-term private key is stored on a hardware device, while a short-term key is stored in memory), one can formally model key exposure by introducing $\mathcal{F}_{\mathsf{Comp}}$ for computations involving the short-term key.

- As a second use case, if we want to study the impact on security that a deployed algorithm has in case it turns out to be unsafe (such as algorithms vulnerable to quantum attacks), this potential attack surface is modeled by formally running the algorithm in question on $\mathcal{F}_{\mathsf{Comp}}$ (without the expectation that this is a special hardware of any sort).

- The abstraction further can capture security assumptions that might be broken at some point. Consider for example the DDH assumption, that is, the assumption that in some group $\mathbb{G} = \langle g \rangle$ random triples $(g^a, g^b, g^c)$ are indistinguishable from DH triples of the form $(g^a, g^b, g^{ab})$ (where the exponents $a$,$b$, and $c$ are picked at random). We can capture the case where the DDH assumption does not hold (anymore) by having the algorithm that generates the DH triple be run inside $\mathcal{F}_{\mathsf{Comp}}$, and model the case that the DDH assumption fails by leaking toward the adversary (via the sub-corrupt command) the randomness used to generate the triples, hence giving a noticeable distinguishing advantage to the adversary.

**Functionality $\mathcal{F}_{\mathsf{Comp}}$**

The functionality maintains a corruption set denoted by $\mathcal{C}$ (initially empty).

1. Upon receiving a command $(\textsc{initialize}, \mathrm{sid}, \{\mathsf{alg}\}_{i\in[\lambda]})$ from an honest party $P$ store $(\{\mathsf{alg}\}_{i\in[\lambda]}, P, \mathrm{sid})$ and send $(\{\mathsf{alg}\}_{i\in[\lambda]}, P)$ to $\mathcal{A}$.
2. Upon receiving a command $(\textsc{query}, \mathrm{sid}, \mathsf{alg}, \mathsf{input})$ from an honest party $P$.
   - If $P \notin \mathcal{C}$ then if the tuple $(\mathsf{alg}, P)$ is stored then compute $\mathsf{output} \leftarrow \mathsf{alg}(\mathsf{input})$, store $(\mathsf{input}, \mathsf{output}, P)$ and return $\mathsf{output}$ to $P$ (note that the input also specifies the random tape for $\mathsf{alg}$).
   - If $P \in \mathcal{C}$ then send $(\mathsf{input}, \mathrm{sid}, P)$ to $\mathcal{A}$.
3. Upon receiving $(\mathsf{output}, P)$ from $\mathcal{A}$, where $P \in \mathcal{C}$ send $\mathsf{output}$ to $P$.
4. Upon receiving $(\textsc{sub-corrupt}, P)$ from $\mathcal{A}$ return all the entries $(\cdot, P)$ stored so far to $\mathcal{A}$ and add $P$ to $\mathcal{C}$.

Figure 1: Functionality $\mathcal{F}_{\mathsf{Comp}}$.

**Corruptions in the ideal world.** As soon as we have a more fine-grained corruption model in the real world, we must reflect it in the ideal world, too as explained in Section 2.2. In particular, the corruption mode is more fine grained and the corruption set $\mathcal{C}$ does contain not only party identities but also particular "(sub-)sessions". Consequently, ideal functionalities can give more fine-grained guarantees depending on which subroutines are corrupted.

More concretely, in the case that the real-world protocol is formulated w.r.t. $\mathcal{F}_{\mathsf{Comp}}$, the ideal functionality accepts inputs $(\textsc{sub-corrupt}, id, P)$ for $id = (\mathcal{F}_{\mathsf{Comp}}, \mathrm{ssid})$ on its backdoor tape and add $(\mathcal{F}_{\mathsf{Comp}}, \mathrm{ssid}, P)$ to $\mathcal{C}$ (and thus reports this extended identity as corrupted). The subsequent behavior of the functionality can thus be dependent on which subroutine is corrupted to give a fine-grained analysis of what happens when which subroutine is corrupted.

In the same spirit, additional modes for further subroutines can be specified. Note that a functionality with a fine-grained corruption model will typically manage subroutine identities that reflect the subroutines of its realizing protocol (e.g., an identity denoting a CRS). Of course, it will always be possible to abstract the naming conventions and impose an ordering of subroutines (to keep the definition of the ideal functionality independent of the exact resources realizing it). However, we stick to the more suggestive notation for clarity.

**Update Systems: healing from sub-corruptions.** For a given functionality $\mathcal{F}$ in the standard corruption model (i.e., maintaining a corruption set consisting of the identifiers of corrupted parties), one can formally extend its behavior to the more detailed case of sub-corruptions by making the effect of sub-corruption equivalent to full corruption: the code of the functionality considers $P$ as corrupted as soon as $P$ gets standard corrupted or there is some entry $(id, P) \in \mathcal{C}$ for some $id$ (which happens if just some subroutine of $P$, like $\mathcal{F}_{\mathsf{Comp}}$, is corrupted). We denote this extension of a standard functionality to the more fine grain setting by $\widehat{\mathcal{F}}$. Based on the above corruption model, the goal of an update system can now be understood in a concise way: an update system should guarantee that even if a party $P$ is sub-corrupted in some instance of, say, functionality $\widehat{\mathcal{F}}$ (and thus can still perform updates as it is not fully corrupted) then after the update to the new functionality $\widehat{\mathcal{F}}$', $P$ should regain its desired security guarantees (until the adversary decides to

sub-corrupt again). Hence, the update is healing the party $P$ and formally corresponds to removing the sub-corruptions entries from the corruption set of the updating protocol instance.

**Example: signatures.** We show two examples for fine-grained corruptions with standard functionalities in Section A for completeness. The standard signature functionality $\mathcal{F}^S_{\mathtt{SIG}}$ for example can be modeled as being subject to sub-corruptions yielding the induced functionality $\widehat{\mathcal{F}}^S_{\mathtt{SIG}}$. As we will see later, the protocol realizing $\widehat{\mathcal{F}}^S_{\mathtt{SIG}}$ is a UC signature protocol as described in Section 2 where the algorithms are executed formally on $\mathcal{F}_{\mathsf{Comp}}$. On the other hand, recall that $\mathcal{F}^S_{\mathtt{SIG}}$ is realized by a signature protocol which is modeled as not being sub-corruptible e.g. to express a different security assumption or more trusted setup. In order to differentiate the two modeling assumptions, we refer to the latter as a *cold signature scheme*. An analogous treatment applies to other cryptographic primitives.

### 3.2 The Update Functionality

We present the update functionality $\mathsf{U}^{\mathscr{F}}_{\mathtt{SttUp},\mathtt{UpPred}}$ in Figure 2 and describe its defining elements here. The update functionality serves as the abstract ideal goal that any real-world update mechanism (defined for a cryptographic primitive or scheme) must satisfy. The functionality is parameterized by three elements: first, by a class of functionalities (or ideal specifications) $\mathscr{F}$ between which transitions (aka updates) are performed, where we demand that there is a dedicated root functionality that we simply call $\mathcal{F}_0$. Second, a function `UpPred`, which, depending on the current state of the system and inputs by parties, decides whether an update is either in progress, failed, or succeeded. And finally, a function `SttUp` which determines the initial state of the new instance of a functionality after a successful update. For any concrete update system, the above three elements fully determine the corresponding ideal update process.

**Basic mode of operation.** $\mathsf{U}^{\mathscr{F}}_{\mathtt{SttUp},\mathtt{UpPred}}$ maintains a rooted graph $\mathsf{UpGraph}$ where each node $v$ stores an instance of an ideal functionality (see Section 2 for the formal definition of an instance). In particular, the values $v.\mathsf{function}$ and $v.\mathrm{ssid}$ determine the code and session of the functionality, respectively (formally, we speak of sub-sessions not to confuse it with the session of the update system as a whole). The root node contains the description of $\mathcal{F}_0$ and its initial state is the initial configuration of $\mathcal{F}_0$. The maintained graph is by construction a directed acyclic graph (DAG) for reasons outlined below. We follow the convention that for two nodes $v$ and $v'$ we call $v'$ a child of $v$ if there is a directed edge from $v$ to $v'$. Likewise, we call $v$ a parent of $v'$ in this case. As we will see, the directed paths away from the root constitute update paths of parties.

At any point in time, every party is assigned to one instance in this DAG structure to which its inputs are relayed toward, and from which it receives the outputs. It is worth to point out that formally following the concept of structured protocols, this makes the actions described in Figure 2 to be *shell-code* (cf. Section 2) as the code deals only with model-related instructions, i.e., how to re-direct inputs and outputs to the correct instances, which are executed as part of the body. The DAG structure models several natural aspects of update systems. For example, a child node $v'$ of $v$ typically contains in $v'.\mathsf{function}$ the description of an updated version of $v.\mathsf{function}$. For example, if $v$ is a leaf of $\mathsf{UpGraph}$ (i.e., does not have children) then $v.\mathsf{function}$ represents an updated functionality that some subset of parties is currently using. In fact, the update functionality will ensure that if parties which start off, say, in the same node $v_i$, agree on the new code, and agree on the same session id, then their inputs and outputs once the update completes are received and returned from the same new instance. On the other hand, the model supports that parties might split and fork into different versions, or to merge previously split parties again into the same, updated session.

**Functionality** $\mathsf{U}^{\mathscr{F}}_{\mathtt{SttUp},\mathtt{UpPred}}$

A rooted, directed acyclic graph $\mathsf{UpGraph}$ is maintained, which is initialized with only the root node $v_0$ with $v_0.\mathsf{function} := \mathcal{F}_0$. The functionality maintains the lists $\mathsf{UpdateReq}_F$ which are lazily created and initially empty. The functionality manages the vector $\mathsf{PartiesFunctions}$. For each $P \in \mathcal{P}$ initially set $\mathsf{PartiesFunctions}[P] \leftarrow v_0$.

*Interaction with most up-to-date functionality per party:*

- If $I := (\textsc{input}, \mathrm{sid}, x)$ is received from a party $P \in \mathcal{P}$ (formally sent via a dummy party that has matching party and session identifiers), set $v \leftarrow \mathsf{PartiesFunctions}[P]$. Then invoke the instance of $v.\mathsf{function}$ on input $(v.\mathrm{ssid}, x)$ from $P$.
- If $I := (\textsc{input-adv}, \mathrm{sid}, x, v)$ is received from the adversary (on the backdoor tape), where $v$ specifies a node of $\mathsf{UpGraph}$ then invoke the instance $v.\mathsf{function}$ on value $(v.\mathrm{ssid}, x)$ on its backdoor tape. Otherwise, ignore the input.
- Upon subroutine-output generation by an instance $v$ that specifies an output value $y$ and destination entity $D$ then output $y$ to $D$. ▷ See Remark 1.
- Upon any other output produced by an instance $v$ destined for the backdoor tape of the adversary, forward the output to the adversary.

*Update Process:*

- If $I := (\textsc{update}, \mathrm{sid}, \mathcal{F})$ for $\mathcal{F} \in \mathscr{F}$ is received from $P$ check that $\mathsf{UpdateStatus}_{P,\mathcal{F}} \neq \mathit{Done}$ and that for all $\mathcal{F}' \neq \mathcal{F} : \mathsf{UpdateStatus}_{P,\mathcal{F}'} \neq \mathit{Updating}$. If the check succeeds then append $(\mathrm{sid}, P)$ to $\mathsf{UpdateReq}_{\mathcal{F}}$ and set $\mathsf{UpdateStatus}_{P,\mathcal{F}} \leftarrow \mathit{Updating}$. Ignore the input if the check fails. Send $(\textsc{update_notification}, \mathrm{sid}, \mathsf{UpdateReq}_{\mathcal{F}}, P, \mathcal{F})$ to $\mathcal{A}$.
- If $I := (\textsc{GrantUpdate}, \mathrm{sid}, P, \mathrm{ssid}, \mathcal{F}', aux)$ for $\mathcal{F}' \in \mathscr{F}$ from the adversary $\mathcal{A}$, then evaluate the predicate $b \leftarrow \mathtt{UpPred}(P, \mathrm{ssid}, \mathcal{F}', \mathsf{UpdateReq}_{\mathcal{F}'}, \mathsf{UpdateStatus}_{P,\mathcal{F}}, \mathsf{PartiesFunctions}, \mathsf{UpGraph}, aux)$ and do the following:
  - If $b = \perp$ then output $(P, \mathrm{ssid}, b = \perp)$ to $\mathcal{A}$.
  - If $b = 0$ then append $(\mathsf{Fail}, P, \mathrm{sid}, \mathrm{ssid})$ to $\mathsf{UpdateReq}_{\mathcal{F}'}$ and set $\mathsf{UpdateStatus}_{P,\mathcal{F}'} \leftarrow \mathit{Done}$. Output $(\mathsf{Fail}, \mathrm{sid}, \mathrm{ssid}, \mathcal{F}')$ to $P$.
  - If $b = 1$ then append $(\mathsf{Success}, P, \mathrm{sid}, \mathrm{ssid})$ to $\mathsf{UpdateReq}_{\mathcal{F}'}$, set $\mathsf{UpdateStatus}_{P,\mathcal{F}'} \leftarrow \mathit{Done}$, and perform the following tasks to complete the update.
    1. Let $v \leftarrow \mathsf{PartiesFunctions}[P]$.
    2. If there is no node $v'$ in $\mathsf{UpGraph}$ with $v'.\mathsf{function} = \mathcal{F}'$ and $v'.\mathrm{ssid} = \mathrm{ssid}$, then create this node as a child of $v$: initialize $v'.\mathsf{function} \leftarrow \mathcal{F}'$, $v'.\mathrm{ssid} \leftarrow \mathrm{ssid}$, and $v'.\mathsf{state} = (\perp, \emptyset)$. Otherwise, let $v'$ be the already existing node and define $v'$ to be a child of $v$ in $\mathsf{UpGraph}$.
    3. Parse $v'.\mathsf{state}$ as $(s, \mathcal{C})$. Assign $\mathsf{PartiesFunctions}[P] \leftarrow v'$ and Compute $\mathsf{state}' \leftarrow \mathtt{SttUp}(v, P, \mathsf{UpdateReq}_{\mathcal{F}'}, \mathsf{PartiesFunctions}, \mathsf{UpGraph}, \mathrm{aux})$, and update $v'.\mathsf{state} \leftarrow (\mathsf{state}', \mathcal{C})$.
    4. Output $(\mathsf{Success}, \mathrm{sid}, \mathrm{ssid}, \mathcal{F}')$ to $P$.

*Corruption:* Upon party corruption of party $P$ via a direct corruption message or via a corruption message to its current instance $\mathsf{PartiesFunctions}[P]$, party $P$ is marked as corrupted and the standard UC corruption mode applies to $\mathsf{U}^{\mathscr{F}}_{\mathtt{SttUp},\mathtt{UpPred}}$. The functionality further reports upon request the aggregated corruption set consisting of all corrupted parties and all reported sub-corruptions (if any) of the instances in $\mathsf{UpGraph}$. ▷ See Section 3.1 for corruption model.

Figure 2: The Generic UC Ideal Update Mechanism.

Furthermore, we demand that in principle, the corruption status of parties is cleared when the new instance starts. This is in essence what captures the basic requirements on a useful update. We note in passing that clearing the corruption status makes only a non-trivial difference when we model sub-corruptions. For full corruptions in the pid-wise corruption model, an attacker can without loss of generality continue to corrupt a party in any instance. Our abstract update therefore captures the requirement that an update system can heal from the less-severe corruptions.

To give an example, if some subset of parties decides to switch to a new functionality, let us call it $\mathcal{F}'$, then we would add a new node $v'$ to $\mathsf{UpGraph}$ as a child of $v$. At a high level, we use $\mathsf{UpGraph}$ to take track of all the updates between instances that occur. The reason why we need to keep track of all the functionalities is that not all the parties might update at the same time. Hence, some parties might still want to use the old functionalities.[1] In the code, $\mathsf{PartiesFunctions}$ is used to store which party is registered to what functionality (i.e., to which node the party is assigned. Any time that a party $P$ queries $\mathsf{U}^{\mathscr{F}}_{\mathtt{SttUp},\mathtt{UpPred}}$ with the command $(\textsc{input}, \mathrm{sid}, x)$, $\mathsf{U}^{\mathscr{F}}_{\mathtt{SttUp},\mathtt{UpPred}}$ derives the actual instance of a program $\mathcal{F}$ this party $P$ is currently talking to and executes the instance of $\mathcal{F}$.

**Updating functionalities and security-relevant parameters.** $\mathsf{U}^{\mathscr{F}}_{\mathtt{SttUp},\mathtt{UpPred}}$ can receive an update command UPDATE from any party $P$ that is willing to *update* to a new specification $\mathcal{F}$. We demand that each party is part of at most one update process by maintaining a flag $\mathsf{UpdateStatus}_{P,\mathcal{F}}$. The command UPDATE comes with an input that specifies the new version. We choose to represent this by the label of the ideal specification for simplicity. If clear from the context, this could also be an index into the functionality class, or any unique label we desire. $\mathsf{U}^{\mathscr{F}}_{\mathtt{SttUp},\mathtt{UpPred}}$ keeps track of all the update requests and of the identifier of the parties that sent the update command by means of a data structure $\mathsf{UpdateReq}$. Whether an update for a party $P$ succeeds depends on a predicate called $\mathtt{UpPred}$. If the adversary admits party $P$ to update, $\mathtt{UpPred}$ decides whether the update can actually be performed. It is therefore the predicate that specifies the ideal conditions for an update to go through (and has to be made concrete in concrete applications).

As said above, the functionality only mandates an abstract update structure such that parties can only be updated along paths in the graph and only if they are willing to update (and, as indicated above, that they have some minimal guarantee whether they are interacting with the same instance. We point out that the predicate also steers whether a party that has been inactive the entire time can *fast-forward* to a more recent version.

Last but not least, core of an update mechanism is to offer a useful form of *state preservation*. For example, with signatures, we have to ensure that legitimate past signatures still verify (in case of an honest signer) whereas the signature scheme is consistent for verifiers. On an abstract level, the state preservation property of an update mechanism for the class $\mathscr{F}$ is given by the function $\mathtt{SttUp}(.)$ which defines how the state of the "new" functionality depends on the ancestor states when an update is made.

In Figure 3 we provide graphical illustration of an execution of $\mathsf{U}^{\mathscr{F}}_{\mathtt{SttUp},\mathtt{UpPred}}$: the initial configuration of $\mathsf{U}^{\mathscr{F}}_{\mathtt{SttUp},\mathtt{UpPred}}$, denoted with $\mathbf{A}$, maintains only one node representing the functionality $\mathcal{F}_0$ and the parties $p_1, \ldots, p_5$ registered to it. From this initial configuration, $p_1$ and $p_2$ update to a new functionality $\mathcal{F}_1$, and a new node is created in the graph. Similarly, the parties $(p_3, p_4)$ update to $\mathcal{F}_2$, thus yielding to the creation of a new node (represented in the configuration $\mathbf{B}$). The adversary then can send any input on the adversarial interface of $\mathcal{F}_2$, in this example we assume that $\mathcal{F}_2$ allows the sub-corruption of the registered parties, and in particular assume that $p_3$ gets corrupted this way. This concludes the description of the configuration we denote with $\mathbf{B}$. We have assumed that all the parties that wanted to update manage to do so, but we recall that it is the update-predicate

[1] We call old functionalities all the functionalities that are not in the leaves of $\mathsf{UpGraph}$.

that mandates whether an update will be successful, depending on the adversarial influence, and the current configuration of the graph. At this point, $p_3$ and $p_4$ may fear that the functionality $\mathcal{F}_2$ has been compromised, and decide to reinitiate $\mathcal{F}_2$. This can be done by letting the parties update to $\mathcal{F}_2$. We note that this creates a new node in the graph with a sub-session identifier $ssid_2' \neq ssid_2$. The state $\mathsf{state}_2'$ depends on $\mathsf{state}_2$, hence some of the work done by the previous instantiation of $\mathcal{F}_2$ may be preserved (how much is preserved is expressed by the state-update function `SttUp`). We stress that in this configuration, denoted with $\mathbf{C}$, $\mathcal{F}_2$ now treats $p_3$ like an honest party, (i.e., it heals from the sub-corruption). In the final configuration (denoted with $\mathbf{D}$) some of the nodes that were initially in $\mathcal{F}_0$ merge again in a new functionality denoted with $\mathcal{F}_3$.

**Abstract properties of update systems.** At this abstract level of defining updates, we can make two natural basic observations: first, a so-called clean install is always possible and corresponds to the case where no state from the past must be preserved and hence is one of the simplest forms of an update. We formally state this in Lemma C.1 in Section C. Second, it is the (amount of) state preservation that steers the complexity and assumptions of any update system. Intuitively, if state preservation is required, some joint view from the past is retained once the update completes and the state-update formalism is precisely the idealization of that real-world process where parties must reach some agreement. While intuitively clear, in Section C we show how one can formalize this claim.

### 3.3 Concluding remarks

We conclude this section on the model with two important observations.

*Remark* 1 (On Subroutine Outputs). As specified in the ideal system in Figure 2, subroutine outputs by functionalities towards some party $P$ are provided to that party. This holds even if the functionality is not the most recent one. We leave it to the higher-level protocol to decide what happens with such outputs by "old" session. Note that one can always define a filter protocol that, based on the sub-session identifier of the most recent functionality, blocks all outputs not belonging to that session. This has the effect that any party only interacts locally with the most recent functionality. We do not hardcode this behavior in the generic update functionality. Observing the old session could turn out useful in settings where a party monitors participation in prior sessions or to detect whether another party is active in two protocol versions. In such cases, further useful actions could be taken thanks to more transparency, which are outside the scope of a pure update protocol.

*Remark* 2. Our DAG based formulation enables the modeling of arbitrary update patterns. We note that even though we do not formally allow cycles in the graph, an update pattern oscillating between functionalities is still easy to capture by assigning unique labels in each invocation of UPDATE. This, for instance, can capture different "ciphersuite" negotiation interactions that occur in the context of key-exchange protocols. Furthermore, multi-parent nodes in the graph open up the opportunity to design more complex update procedures where a set of parties initially diverges from a single functionality and subsequently converges to a single functionality. Such modeling capability can come handy if one wishes to capture complex multiparty settings such as that of a distributed ledger where parties may diverge due to an implementation difference and subsequently negotiate the merging of their states as part of a "convergence" update. We note that studying such complex mechanisms is beyond the scope of our current exposition — nevertheless we believe it is essential that an update framework should be capable to facilitate them.

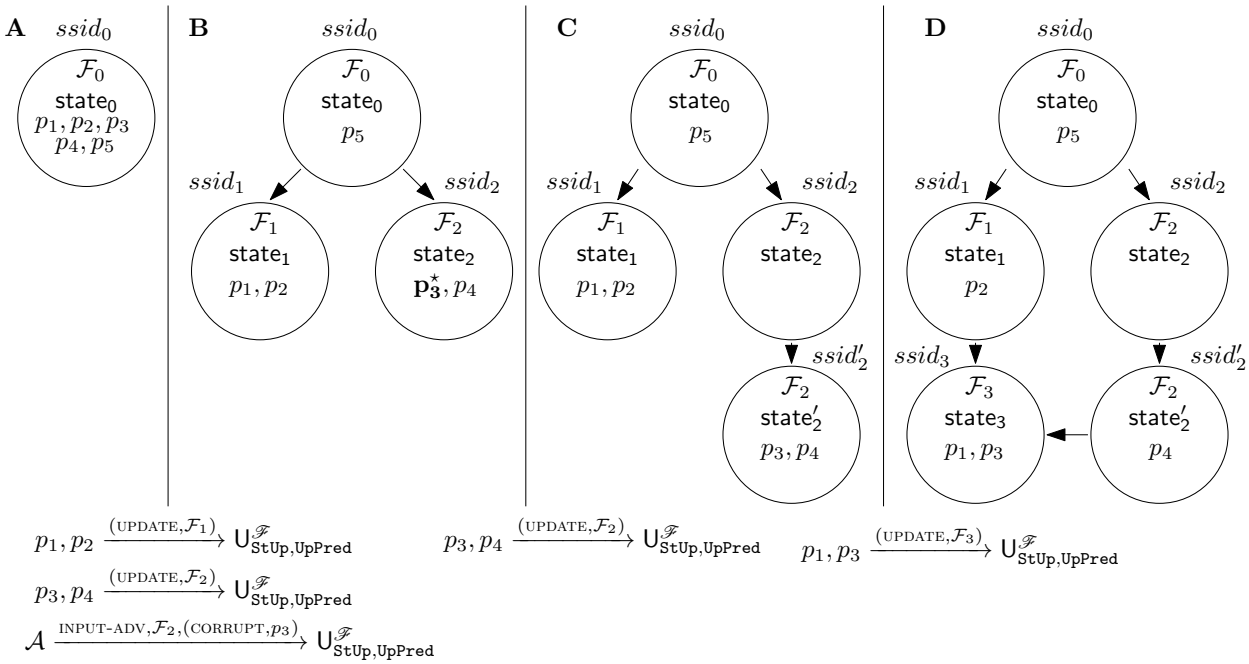


Figure 3: Sample execution of $\mathsf{U}^{\mathscr{F}}_{\mathtt{SttUp},\mathtt{UpPred}}$.

# 4 Updatable Signatures

**Ideal signatures under subroutine corruptions.** Let $\widehat{\mathcal{F}}^S_{\mathtt{SIG}}$ be the signature functionality supporting sub-routine corruptions (i.e., it behaves exactly like the signature functionality but it allows for sub-corruption as described in Section 3.1, see Section A for the formal definition). In this section, we design the protocol $\widehat{\Pi}^S_{\mathcal{DS}}$ (Figure 4) which realizes $\widehat{\mathcal{F}}^S_{\mathtt{SIG}}$ relying on $\mathcal{F}_{\mathsf{Comp}}$. The following statement is immediate:

**Lemma 4.1.** *If $\Pi^S_{\mathcal{DS}}$ realizes $\mathcal{F}_{\mathtt{SIG}}$ (Figure 11) then $\widehat{\Pi}^S_{\mathcal{DS}}$ realizes $\widehat{\mathcal{F}}^S_{\mathtt{SIG}}$ in the $\mathcal{F}_{\mathsf{Comp}}$-hybrid model.*

Recall from Section 3.1 that a scheme is modeled using $\mathcal{F}_{\mathsf{Comp}}$ to indicate the belief that the honest usage can turn out to be unsafe. Recall the distinction to a signature scheme that we assume to actually be safe under honest usage in that it realizes $\mathcal{F}_{\mathtt{SIG}}$ (and thus subroutine corruptions of an honest party are not a concern), which are referred to as cold schemes.

*Remark* 3. In our update mechanisms, we make use of cold vs. hot signature keys as a way to recover from sub-corruptions across updates and to reflect different trust assumptions. We point out that our security goal necessitates a form of "authenticated communication" post sub-corruption and is enabled by this cold signature mechanism. We make that choice due to two reasons: first, it is a practical assumption (in particular when accessing cold keys only rarely or hardware tokens are assumed) and second, because it allows us to describe a generic approach to post-quantum readiness as described in Section 4.3.

## 4.1 Ideal Updatable Signature System

We want to provide a scheme that implements the update system for signatures. More formally, we describe a protocol to UC-realize $\mathsf{U}^{\mathscr{F}_{\mathtt{SIG}}}_{\mathtt{SttUp}_{\mathtt{SIG}},\mathtt{UpPred}_{\mathtt{SIG}}}$ for $\mathscr{F}_{\mathtt{SIG}} := \{\widehat{\mathcal{F}}^{S,\mathcal{DS}_i}_{\mathtt{SIG}}\}_i$, where $\mathscr{F}_{\mathtt{SIG}}$ denotes the

**Protocol $\widehat{\Pi}^S_{\mathcal{DS}}$**

*Initialization*

Let $\mathcal{DS} = (\mathsf{KGen}, \mathsf{Sign}, \mathsf{Ver})$ be the signature scheme. Each party $P \in \mathcal{P}$ sends to $\mathcal{F}_{\mathsf{Comp}}$ the input $(\textsc{initialize}, \mathrm{sid}, \{\mathsf{KGen}, \mathsf{Sign}, \mathsf{Ver}\})$.

*Key Generation:*

Upon receiving the command $I := (\textsc{input}, \mathrm{sid}, \textsc{key-gen})$ $S$ sends $(\textsc{query}, \mathrm{sid}, \mathsf{KGen}, 1^\lambda)$ to $\mathcal{F}_{\mathsf{Comp}}$ and upon receiving $(vk, sk)$ outputs $(\textsc{verification-key}, \mathrm{sid}, vk)$.

*Signing and Verification:*

- Upon receiving the command $I := (\textsc{sign}, m)$ the signer send $(\textsc{query}, \mathrm{sid}, \mathsf{KGen}, (sk, m))$ to $\mathcal{F}_{\mathsf{Comp}}$. Upon receiving $(m, \sigma)$ from $\mathcal{F}_{\mathsf{Comp}}$ output $(\textsc{signature}, \mathrm{sid}, m, \sigma)$.
- Upon receiving the command $I = (\textsc{verify}, vk, m, \sigma)$ send $(\textsc{query}, \mathrm{sid}, \mathsf{Ver}, (vk, m, \sigma))$ to $\mathcal{F}_{\mathsf{Comp}}$, and upon receiving a reply $b$ output $(\textsc{verified}, \mathrm{sid}, m, b)$.

Figure 4: Corruptible signature

family of signature functionalities as defined above which are formally indexed with $\mathcal{DS}_i$. Note that $S$ (the signer identity) appears as an explicit identifier in this family.

To formalize the ideal update for signatures, we first define how $\mathtt{SttUp}_{\mathtt{SIG}}$ and $\mathtt{UpPred}_{\mathtt{SIG}}$ work. The state $\mathsf{state}_{\mathtt{SIG}}$ of a signature functionality (aside of the corruption set) is represented by the verification key of the sender and the set of tuples $(vk, m, \sigma, b)$ stored by the functionality (we refer to Section A for a more detailed discussion).

- $\mathtt{UpPred}_{\mathtt{SIG}}$: On input $(P, \mathrm{ssid}, \mathcal{F}', \mathsf{UpdateReq}_{\mathcal{F}'}, \mathsf{UpdateStatus}_{P,\mathcal{F}'}, \mathsf{PartiesFunctions}, \mathsf{UpGraph}, 0^\lambda)$ do the following: if $P$ appears in $\mathsf{UpdateReq}$ and $\mathsf{UpdateStatus}_{P,\mathcal{F}'} = \mathit{Updating}$ then let $v \leftarrow \mathsf{PartiesFunctions}[P]$ and perform the following steps to compute the decision (otherwise, return $\perp$):
  - If $P$ is the signer of instance $v$ and $\mathsf{UpGraph}$ has no node $v'$ with $v'.\mathsf{function} = \mathcal{F}'$ and $v'.\mathrm{ssid} = \mathrm{ssid}$ then return 1.
  - Else, if $P$ is not the signer and if there is a node $v'$ with $v'.\mathsf{function} = \mathcal{F}'$ and $v'.\mathrm{ssid} = \mathrm{ssid}$ that is a child of $v$ and $(S, vk) \in v'.\mathsf{state}$ then do the following.
    - If $S$ is honest then return 1.
    - If $S$ is corrupted then let $M_P$ be the set of all the pairs $(m, \sigma)$ s.t. $(P, vk, m, \sigma, 1) \in v.\mathsf{state}$ and let $M$ be the set of all the pairs $(m, \sigma)$ for which a tuple $(\cdot, vk, m, \sigma, 1)$ is stored in $v'.\mathsf{state}$, if $M_P \subseteq M$ then return 1.
  - Return 0.

The update predicate thus ensures that it only allows a party to transition, if it can be guaranteed that the party will remain consistent with respect to the set of messages it has seen and successfully verified. Furthermore, as long as the signer is honest, the update graph $\mathsf{UpGraph}$ is a chain of instances of the signature functionality. We next define the state updates:

- $\mathtt{SttUp}_{\mathtt{SIG}}$: On input $\mathtt{SttUp}(v^\star, P, \mathsf{UpdateReq}, \mathsf{PartiesFunctions}, \mathsf{UpGraph}, \mathrm{aux})$ first obtain the instance $v \leftarrow \mathsf{PartiesFunctions}[P]$ (note that by definition, $v^\star$ is the parent node of $v$ party $P$ is updating from):

- If $P$ is the signer in instance $v^\star$ and $P$ is honest, define the new state $s$ and insert the tuple $(P, vk) \in v^\star.\mathsf{state}$ as well as all tuples $(S, vk, m, \sigma, 1) \in v^\star.\mathsf{state}$.
- Else, if $P$ is the signer for the instance $v^\star$ and $P$ is corrupted then define an empty state $s$, and set $s \leftarrow aux$.
- If $P$ is not the signer then set $s \leftarrow v.\mathsf{state}$.
- Return $s$.

The definition means that we keep all generated signatures of the honest signer, even if the signer has been sub-corrupted. If instead the signer is fully corrupted, then we let the adversary decide the state of the updated functionality.

## 4.2 The Protocol and Security Statement

The main underlying primitives that we use to realize our update system are a cold signature scheme $\mathcal{DS}_c$ and a message registry that we formally introduce in Section D. At a very high level, $\mathcal{DS}_c$ will be used only during the update phase by the signer to authenticate the information that concerns the updates which are sent to each individual verifier. We prove that as long as the security of $\mathcal{DS}_c$ holds then our scheme realizes the updatable functionality $\mathsf{U}^{\mathscr{F}_{\mathtt{SIG}}}_{\mathtt{SttUp}_{\mathtt{SIG}},\mathtt{UpPred}_{\mathtt{SIG}}}$. Our approach is justified by the fact that $\mathcal{DS}_c$ might be a particularly inefficient signature scheme, but at the time difficult to attack. Given the inefficiency of $\mathcal{DS}_c$, it might not be practical to directly use it as a signature scheme, hence we use it just to support an update between one signature scheme and another (which are more efficient than $\mathcal{DS}_c$ but they might be compromised more easily). In order to provide the security guarantees we modularize our construction with a simple cryptographic primitive we call a message registry. The primitive uses three algorithms $(\mathsf{Gen}, \mathsf{Add}, \mathsf{Vmr})$ and enables to deposit messages to a "container", a process that may generate a witness, while given a message and witness one can verify reliably whether the message is in the container. A message registry can be trivially realized information theoretically, or cryptographically for more efficiency. We give more details in Appendix D.

Equipped with these tools, our update system works as follows. The signer registers his cold verification-key to a PKI[2], and starts issuing signatures using the (sub-corruptible) signature functionality $\widehat{\mathcal{F}}^{S,\mathcal{DS}_0}_{\mathtt{SIG}} \in \mathscr{F}_{\mathtt{SIG}}$ while the verifier simply uses the same functionality to verify the signatures. The signer keeps the memory of all the signatures he issues (by recording them in the message registry), and the verifiers do something similar, i.e., memorizing valid signatures.

When the signer wants to update, he hashes the message registry, together with the description of the new functionality he wants to update to (let us call it $\widehat{\mathcal{F}}^{S,\mathcal{DS}_\star}_{\mathtt{SIG}}$) and a verification key freshly generated by querying $\widehat{\mathcal{F}}^{S,\mathcal{DS}_\star}_{\mathtt{SIG}}$. Then he signs the hash with the cold signature scheme and sends the signature, the message registry, $\widehat{\mathcal{F}}^{S,\mathcal{DS}_\star}_{\mathtt{SIG}}$ and the new verification key to each verifier.

If a verifier has received an update command for the functionality $\widehat{\mathcal{F}}^{S,\mathcal{DS}_\star}_{\mathtt{SIG}}$ (which is the same functionality he receives from the signer, then), and if all the information are authenticated as described above, then the verifier does the following. He checks whether all the signatures he has ever verified are contained in the message registry received from the signer, and if this is the case then the verifier will accept the update. The reason for this check is to maintain consistency throughout the updates. That is, if a verifier has verified successfully a signature for a message $m$, then any verifier registered to the same functionality must correctly verify the same pair, even if the verification of $m$ happened in a previous epoch.

[2]This enables authenticated communication from the honest signer to other parties. In the protocol, we also store the initial verification key of the signer for efficiency and simplicity, as otherwise, the first update message would have to include it.

To fully enable consistency, we also need to modify how the verification procedure works after an update. The verifier now, upon receiving a message and a signature, would first check if the pair is valid with respect to the new key by querying $\widehat{\mathcal{F}}_{\mathtt{SIG}}^{S,\mathcal{DS}_\star}$. If it is, then the verifier memorizes the pair as before, if instead $\widehat{\mathcal{F}}_{\mathtt{SIG}}^{S,\mathcal{DS}_\star}$ returns 0, then the verifier looks in the message registry (received during the update) for the pair. If there is a match, the verifier outputs 1, else he returns 0.

The overall idea is simple, but we need to deal with some subtle technicalities to be able to prove that the scheme does realize $\mathsf{U}_{\mathtt{SttUp}_{\mathtt{SIG}},\mathtt{UpPred}_{\mathtt{SIG}}}^{\mathscr{F}_{\mathtt{SIG}}}$. The first is related to the fact that in $\mathsf{U}_{\mathtt{SttUp}_{\mathtt{SIG}},\mathtt{UpPred}_{\mathtt{SIG}}}^{\mathscr{F}_{\mathtt{SIG}}}$ the sub-session identifiers play an important role since they define to which node in the graph (maintained by $\mathsf{U}_{\mathtt{SttUp}_{\mathtt{SIG}},\mathtt{UpPred}_{\mathtt{SIG}}}^{\mathscr{F}_{\mathtt{SIG}}}$) each party should be registered to. In the real world protocol the ssid is computed by hashing the descriptor of $\widehat{\mathcal{F}}_{\mathtt{SIG}}^{S,\mathcal{DS}_\star}$ with the message registry the signer sends during the update. The idea is that we want verifiers with the same sub-session identifier to share the same history of signed messages (i.e., they are registered to the same functionality with the same state). The collision resistance of the hash function guarantees that if two verifiers accept an update for the same ssid, then they must have received the same message-registry and the same functionality descriptor from the adversary, creating the correct connection between ideal and real world. A final technicality is that we must keep the honest signer key the same across epochs to be compatible with standard signatures (this is an important aspect looking ahead on Section 4.4). The full description of the protocol is given in Figures 5 and 6.

*Remark* 4. It is instructive to compare the above construction to a simple construction that works as follows: during the update, simply sign the message set with the next instance of the signature public key. There are two drawbacks of this update system compared to our proposal: first, the party has to update its record with the PKI (and hence revoke the old one as it might become compromised later). Second, while the system meets the minimal requirement that we have uninterrupted security as long as the security breach of a scheme always happens "after an update" to the next version, the simple construction falls short in enabling the honest signer to regain security after a security breach of the current implementation. In particular, our $\mathtt{SttUp}_{\mathtt{SIG}}$ demands that an honest signer (despite being subcorrupted) regains full security in the updated session, while for the above simpler protocol, the update guarantees are lost when the active key is compromised (since the most recent update messages can be equivocated by an attacker). Note that an even stronger PKI with a richer interface could salvage the scheme (in spirit along the lines of our construction), but assuming such a strong PKI does not appear to be a practical assumption. In contrast, our scheme works with the basic CA-functionality of Canetti [Can03] and hence with the minimal root of trust (just a single key per signer must be distributed authentically in the beginning), which is easily seen to be a necessary assumption to achieve any non-trivial form of state preservation. We elaborate on this crucial aspect when we discuss the benefits of our construction in the larger context of post-quantum readiness in Section 4.3.

**Theorem 4.2.** *Assume* $(\mathsf{Gen}_{\mathsf{H}}, H)$ *is a collision-resistant hash function,* $\mathcal{DS}_c$ *is a (cold) signature scheme which is only used during the update, and* $(\mathsf{Gen}, \mathsf{Add}, \mathsf{Vmr})$ *is a Message Registry then* $\Pi_{\mathtt{SIG}}^{S,up}$ *realizes* $\mathsf{U}_{\mathtt{SttUp}_{\mathtt{SIG}},\mathtt{UpPred}_{\mathtt{SIG}}}^{\mathscr{F}_{\mathtt{SIG}}}$ *for the function family defined above in the standard PKI hybrid model.*

*Proof.* Let $\mathcal{A}$ be an arbitrary polynomial-time adversary. We will describe a corresponding polynomial time ideal process adversary $\mathcal{S}$ such that no non-uniform polynomial-time environment can distinguish whether $\Pi_{\mathtt{SIG}}^{S,up}$ is running in the real-world with parties $p_1, \ldots, p_n$ and the adversary $\mathcal{A}$ or in the ideal process running with $\mathsf{U}_{\mathtt{SttUp}_{\mathtt{SIG}},\mathtt{UpPred}_{\mathtt{SIG}}}^{\mathscr{F}_{\mathtt{SIG}}}$, $\mathcal{S}$ and the dummy parties $\tilde{p}_1, \ldots, \tilde{p}_n$.

We assume that $\mathsf{U}_{\mathtt{SttUp}_{\mathtt{SIG}},\mathtt{UpPred}_{\mathtt{SIG}}}^{\mathscr{F}_{\mathtt{SIG}}}$ is initialized with the root $v_0$ with $v_0.\mathrm{ssid} = 0$ and $v_0.\mathsf{function} = \widehat{\mathcal{F}}_{\mathtt{SIG}}^{S,\mathcal{DS}_0}$. The simulator $\mathcal{S}$ maintains an initially empty set of sub-corrupted parties $\mathsf{SubC}$ the list

**Protocol $\Pi^{S,up}_{\mathtt{SIG}}$ - Part I**

*Initialization*

All the parties initially interact to the root functionality $\widehat{\mathcal{F}}^{S,\mathcal{DS}_0}_{\mathtt{SIG}} \in \mathscr{F}_{\mathtt{SIG}}$. Each party $P \in \mathcal{P}$ initializes $\mathtt{Version}^P \leftarrow 0$, $\mathrm{sid}^P \leftarrow \mathrm{sid}$ (where sid is the session-ID of the protocol instance), $M^P \leftarrow \emptyset$, and defines $\mathsf{Ver}^P$ which on input $(vk, m, (\sigma, w))$ sends $(\textsc{verify}, \mathrm{sid}^P, vk, m, \sigma)$ to $\widehat{\mathcal{F}}^{S,\mathcal{DS}_0}_{\mathtt{SIG}}$ and returns whatever $\widehat{\mathcal{F}}^{S,\mathcal{DS}_0}_{\mathtt{SIG}}$ returns.

*Key Generation:*

Upon receiving the command $I := (\textsc{input}, \mathrm{sid}, \textsc{key-gen})$ $S$ does the following.

- Send $(\textsc{key-gen}, \mathrm{sid}^S)$ to $\widehat{\mathcal{F}}^{S,\mathcal{DS}_0}_{\mathtt{SIG}}$ thus obtaining $vk_0$.
- Compute $(vk_c, sk_c) \leftarrow \mathsf{KGen}_c(1^\lambda)$ and $k \xleftarrow{\$} \mathsf{Gen}_{\mathsf{H}}(1^\lambda)$
- Register $(vk_0, vk_c, k)$ with the PKI.
- Generate $trap, A \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$ and define $\mathsf{Sign}$ which is a stateful algorithm that on input $m$ does the following steps.
  - Send $(\textsc{sign}, m)$ to $\widehat{\mathcal{F}}^{S,\mathcal{DS}_0}_{\mathtt{SIG}}$ and upon receiving $(\textsc{signature}, \mathrm{sid}, m, \sigma)$ continue.
  - Compute $A', w \xleftarrow{\$} \mathsf{Add}(trap, A, (m, vk_0, \sigma))$.
  - Set $A \leftarrow A'$ and return $(m, (\sigma, w))$.

Return $(\textsc{verification-key}, \mathrm{sid}, vk_0)$.

*Signing and Verification:*

- Upon receiving the command $I := (\textsc{input}, \mathrm{sid}, y)$,
  - if $y = (\textsc{sign}, m)$ and $I$ is received by the signer, then run $\mathsf{Sign}(m)$ and return whatever $\mathsf{Sign}$ returns.
  - if $y = (\textsc{verify}, vk, m, (\sigma, w))$ and $I$ is received by a party $P$ then run $\mathsf{Ver}^P$ on input $(vk, m, (\sigma, w))$ to receive $(\textsc{verified}, \mathrm{sid}, m, b)$; add $(m, vk, (\sigma, w))$ to $M^P$ if $b = 1$, and finally return $(\textsc{verified}, \mathrm{sid}, m, b)$.

Figure 5: Updatable signatures: initialization and operation.

$\mathsf{pNodes}$ (that maintains the association between parties, sub-session identities and verification keys) initialized with $\mathsf{pNodes}[p_i] \leftarrow (0, \widehat{\mathcal{F}}^{S,\mathcal{DS}_0}_{\mathtt{SIG}}, vk_0)$ for all $p_i \in \mathcal{P}$. $\mathcal{S}$ works as follows.

- All the honest parties simulated by $\mathcal{S}$ in the real world are initialized as described in $\Pi^{S,up}_{\mathtt{SIG}}$.
- If $p_i$ receives $(\textsc{update_notification}, \mathsf{UpdateReq}, p_i, \widehat{\mathcal{F}}^{S,\mathcal{DS}_\star}_{\mathtt{SIG}})$ from $\mathsf{U}^{\mathscr{F}_{\mathtt{SIG}}}_{\mathtt{SttUp}_{\mathtt{SIG}},\mathtt{UpPred}_{\mathtt{SIG}}}$ and $p_i$ is a verifier then add $(p_i, \widehat{\mathcal{F}}^{S,\mathcal{DS}_\star}_{\mathtt{SIG}})$ to $\mathsf{waiting}$; if $p_i$ is an honest signer then set $\mathrm{sid}^{p_i} \leftarrow H_k(A||\widehat{\mathcal{F}}^{S,\mathcal{DS}_\star}_{\mathtt{SIG}})$ and do the following
  1. Send $(\textsc{key-gen}, \mathrm{sid}^{p_i})$ to $\widehat{\mathcal{F}}^{S,\mathcal{DS}_\star}_{\mathtt{SIG}}$ (the simulator internally runs $\widehat{\mathcal{F}}^{S,\mathcal{DS}_\star}_{\mathtt{SIG}}$) thus obtaining $(vk_\star, sk_\star)$.
  2. Set $\mathtt{Version} \leftarrow \mathtt{Version} + 1$.
  3. Compute $\sigma_c \leftarrow \mathsf{Sign}_c(sk_c, \mathtt{Version}||vk_\star||h)$, define $\tau \leftarrow \widehat{\mathcal{F}}^{S,\mathcal{DS}_\star}_{\mathtt{SIG}}||\mathtt{Version}||vk_\star||A||h||\sigma_c$ and send $\tau$ to each verifier.
  4. Set $\mathsf{pNodes}[p_i] \leftarrow (\mathrm{sid}^{p_i}, \widehat{\mathcal{F}}^{S,\mathcal{DS}_\star}_{\mathtt{SIG}}, vk_\star)$ and send $(\textsc{GrantUpdate}, \mathrm{sid}, p_i, \mathrm{sid}^{p_i}, \widehat{\mathcal{F}}^{S,\mathcal{DS}_\star}_{\mathtt{SIG}}, 0^\lambda)$

**Protocol $\Pi^{S,up}_{\mathtt{SIG}}$ - Part II**

*Update:*

- If the signer $S$ receives the command $I := (\textsc{update}, \mathrm{sid}, \widehat{\mathcal{F}}^{S,\mathcal{DS}_\star}_{\mathtt{SIG}})$ (with $\widehat{\mathcal{F}}^{S,\mathcal{DS}_\star}_{\mathtt{SIG}} \in \mathscr{F}_{\mathtt{SIG}}$ then she sets $\mathrm{sid}^S \leftarrow H_k(A||\widehat{\mathcal{F}}^{S,\mathcal{DS}_\star}_{\mathtt{SIG}})$ does the following steps.

  1: Send $(\textsc{key-gen}, \mathrm{sid}^S)$ to $\widehat{\mathcal{F}}^{S,\mathcal{DS}_\star}_{\mathtt{SIG}}$ thus obtaining $vk_\star$.

  2: Redefine $\mathsf{Sign}$. On input $m$, $\mathsf{Sign}$ does the following steps.
  - Send $(\textsc{sign}, \mathrm{sid}^S, m)$ to $\widehat{\mathcal{F}}^{S,\mathcal{DS}_\star}_{\mathtt{SIG}}$ and upon receiving $(\textsc{signature}, \mathrm{sid}^S, m, \sigma)$ continue.
  - Compute $A', w \xleftarrow{\$} \mathsf{Add}(trap, A, (m, vk_0, \sigma))$.
  - Set $A \leftarrow A'$ and return $(m, (\sigma, w))$.

  3: Set $\mathtt{Version} \leftarrow \mathtt{Version} + 1$.

  4: Compute $\sigma_c \leftarrow \mathsf{Sign}_c(sk_c, \mathtt{Version}||vk_\star||h)$, define $\tau \leftarrow \widehat{\mathcal{F}}^{S,\mathcal{DS}_\star}_{\mathtt{SIG}}||\mathtt{Version}||vk_\star||A||h||\sigma_c$ and send $\tau$ to each verifier.

- If a party $P$ receives the command $I := (\textsc{update}, \mathrm{sid}, \widehat{\mathcal{F}}^{S,\mathcal{DS}_\star}_{\mathtt{SIG}})$, if $\widehat{\mathcal{F}}^{S,\mathcal{DS}_\star}_{\mathtt{SIG}} \in \mathscr{F}_{\mathtt{SIG}}$ then she does the following steps.

  1: Wait to receive $\tau$ from the signer $S$. Once the message $\tau$ is received, first retrieve the signer's public keys and hash key $(vk_0, vk_c, k)$ from the PKI. Then parse $\tau$ as $\mathcal{F}||\mathtt{Version}||vk_\star||A||h||\sigma_c$, check if $\mathsf{Ver}_c(vk_c, \mathtt{Version}||vk_\star||h, \sigma_c) = 1$ and $h = H_k(A||\mathcal{F})$ and $\mathcal{F} = \widehat{\mathcal{F}}^{S,\mathcal{DS}_\star}_{\mathtt{SIG}}$ and $\mathtt{Version} > \mathtt{Version}^P$ and $\wedge_{(m,vk_0,(\sigma,w)) \in M^P} \mathsf{Vmr}(A, w, (m, \sigma, vk_0))$. If the check fails then ignore the input, else set $A^P \leftarrow A$, $\mathrm{sid}^P \leftarrow h$, set $\mathtt{Version}^P \leftarrow \mathtt{Version}$ and redefine $\mathsf{Ver}^P$ as follows.
  $\mathsf{Ver}^P(vk', m, (\sigma, w))$:
  - If $vk' = vk_0$, then send $(\textsc{verify}, \mathrm{sid}^P, vk_\star, m, \sigma)$ to $\widehat{\mathcal{F}}^{S,\mathcal{DS}_\star}_{\mathtt{SIG}}$; upon receiving $(\textsc{verified}, \mathrm{sid}^P, m, b)$, if $b = 1$ then append $(m, vk_0, (\sigma, w))$ to $M^P$ and return 1. If $b = 0$ then return $\mathsf{Vmr}(A, w, (m, \sigma, vk_0))$.
  - else return 0.

Figure 6: Updatable signatures: update step

- Let $p_i$ be an honest real-world verifier simulated by $\mathcal{S}$. If $p_i$ receives $\tau$ from a corrupted signer, then do the following
  - Parse $\tau$ as $\mathcal{F}||\mathtt{Version}||vk_\star||A||h||\sigma_c$ and send $I := (\textsc{update}, \mathrm{sid}, \mathcal{F})$ to $\mathsf{U}^{\mathscr{F}_{\mathtt{SIG}}}_{\mathtt{SttUp}_{\mathtt{SIG}},\mathtt{UpPred}_{\mathtt{SIG}}}$ on the behalf of the corrupted signer.
  - Upon receiving $(\textsc{update_notification}, \mathsf{UpdateReq}, S, \mathcal{F})$ from $\mathsf{U}^{\mathscr{F}_{\mathtt{SIG}}}_{\mathtt{SttUp}_{\mathtt{SIG}},\mathtt{UpPred}_{\mathtt{SIG}}}$ where $S$ is the corrupted signer do the following.
    * Compute $\mathrm{ssid} \leftarrow H_k(A||\widehat{\mathcal{F}}^{S,\mathcal{DS}_\star}_{\mathtt{SIG}})$
    * Send $I := (\textsc{GrantUpdate}, \mathrm{sid}, S, \mathrm{ssid}, \mathcal{F}, 0^\lambda)$ to $\mathsf{U}^{\mathscr{F}_{\mathtt{SIG}}}_{\mathtt{SttUp}_{\mathtt{SIG}},\mathtt{UpPred}_{\mathtt{SIG}}}$.
  - If there is an entry $(p_i, \widehat{\mathcal{F}}^{S,\mathcal{DS}_\star}_{\mathtt{SIG}})$ in $\mathsf{waiting}$ then do the following else stop.
    1. Check if $\mathsf{Ver}_c(vk_c, \mathtt{Version}||vk_\star||h, \sigma_c) = 1$ and $h = H_k(A||\mathcal{F})$ and $\mathcal{F} = \widehat{\mathcal{F}}^{S,\mathcal{DS}_\star}_{\mathtt{SIG}}$ and $\mathtt{Version} > \mathtt{Version}^{p_i}$ and $\wedge_{(m,vk,w) \in M_{p_i}} \mathsf{Vmr}(A, w, (m, vk))$. If the check fails then ignore the input, else set $A^{p_i} \leftarrow A$, $\mathrm{sid}^{p_i} \leftarrow h$, set $\mathtt{Version}^{p_i} \leftarrow \mathtt{Version}$.
    2. Set $\mathsf{pNodes}[p_i] \leftarrow (\mathrm{sid}^{p_i}, \widehat{\mathcal{F}}^{S,\mathcal{DS}_\star}_{\mathtt{SIG}}, vk_\star)$ and send $(\textsc{GrantUpdate}, \mathrm{sid}, p_i, \mathrm{sid}^{p_i}, \widehat{\mathcal{F}}^{S,\mathcal{DS}_\star}_{\mathtt{SIG}}, 0^\lambda)$ to $\mathsf{U}^{\mathscr{F}_{\mathtt{SIG}}}_{\mathtt{SttUp}_{\mathtt{SIG}},\mathtt{UpPred}_{\mathtt{SIG}}}$

- If the signer $S$ receives $(\textsc{sign}, \mathrm{sid}, m)$ from $\mathsf{U}^{\mathscr{F}_{\mathtt{SIG}}}_{\mathtt{SttUp}_{\mathtt{SIG}},\mathtt{UpPred}_{\mathtt{SIG}}}$ then set $(\mathrm{sid}^S, \mathcal{F}, vk) \leftarrow \mathsf{pNodes}[S]$ send $(\textsc{sign}, \mathrm{sid}^S, m)$ to $\mathcal{F}$. Upon receiving $I = (\textsc{signature}, \mathrm{sid}^S, m, \sigma)$ from $\mathcal{F}$ do the following
  - Compute $A', w \xleftarrow{\$} \mathsf{Add}(trap, A_S, (m, vk_0, \sigma))$ and set $A_S \leftarrow A'_S$.
  - Forward $I = (\textsc{signature}, \mathrm{sid}^S, m, (\sigma, w))$ to $\mathsf{U}^{\mathscr{F}_{\mathtt{SIG}}}_{\mathtt{SttUp}_{\mathtt{SIG}},\mathtt{UpPred}_{\mathtt{SIG}}}$.
- If the party $p_i$ receives $I = (\textsc{verify}, \mathrm{sid}, m, (\sigma, w), vk')$ from $\mathsf{U}^{\mathscr{F}_{\mathtt{SIG}}}_{\mathtt{SttUp}_{\mathtt{SIG}},\mathtt{UpPred}_{\mathtt{SIG}}}$ then set $(\mathrm{sid}^S, \mathcal{F}, vk) \leftarrow \mathsf{pNodes}[p_i]$, send $I = (\textsc{verify}, \mathrm{sid}^{p_i}, m, \sigma, vk)$ to $\mathcal{F}$. Upon receiving $(\textsc{verified}, \mathrm{sid}, m, b)$, if $b = 1$ then append $(m, vk_0, \sigma)$ to $M^{p_1}$ and return $(\textsc{verified}, \mathrm{sid}, m, 1)$. If $b = 0$ then set $d \leftarrow \mathsf{Vmr}(A_{p_i}, w, (m, \sigma, vk_0))$ and return $(\textsc{verified}, \mathrm{sid}, m, 1)$ to $\mathsf{U}^{\mathscr{F}_{\mathtt{SIG}}}_{\mathtt{SttUp}_{\mathtt{SIG}},\mathtt{UpPred}_{\mathtt{SIG}}}$
- If the party $p_i$ receives any command $I = (\ldots, \mathrm{sid}, \ldots)$ from $\mathsf{U}^{\mathscr{F}_{\mathtt{SIG}}}_{\mathtt{SttUp}_{\mathtt{SIG}},\mathtt{UpPred}_{\mathtt{SIG}}}$ the set $(\mathrm{sid}^S, \mathcal{F}, vk) \leftarrow \mathsf{pNodes}[p_i]$, send $I$ to $\mathcal{F}$ and forward the reply obtained from the functionality to $\mathsf{U}^{\mathscr{F}_{\mathtt{SIG}}}_{\mathtt{SttUp}_{\mathtt{SIG}},\mathtt{UpPred}_{\mathtt{SIG}}}$.

We note that there are only three reasons why the simulation could fail: 1) After the update to a functionality $\widehat{\mathcal{F}}^{S,\mathcal{DS}_\star}_{\mathtt{SIG}}$ with sub-session-id ssid, the state of a functionality (which is maintained by $\mathsf{U}^{\mathscr{F}_{\mathtt{SIG}}}_{\mathtt{SttUp}_{\mathtt{SIG}},\mathtt{UpPred}_{\mathtt{SIG}}}$) does not contain a tuple $(m, (\sigma, w))$ that is instead contained message registry $A$ of an honest verifier that accepted to update to $\widehat{\mathcal{F}}^{S,\mathcal{DS}_\star}_{\mathtt{SIG}}$ with sub-session-id ssid; 2) Two (or more) honest verifiers $(p_1, p_2)$ accepted in the real world to update to a functionality with sub-session-id ssid but after the update the two verifiers accept two different message registries; 3) In the real world a verifier accepted to update to a functionality with sub-session-id ssid for which the signer is honest, but this functionality (with the same ssid) does not appear in the graph maintained by $\mathsf{U}^{\mathscr{F}_{\mathtt{SIG}}}_{\mathtt{SttUp}_{\mathtt{SIG}},\mathtt{UpPred}_{\mathtt{SIG}}}$ in the ideal world.

Case 1) implies that the message registry $A$ of an honest verifier contains a tuple $T := (m, vk, \sigma)$ but the honest signer has never added $T$ to the message registry he sent during the update. However, this can happen only with negligible probability as otherwise, we would be able to contradict the security of the message registry. If case 3) happens it must be because $H_k(A||\mathcal{F}) = H_k(A'||\mathcal{F})$ where $A \neq A'$. Hence, this even can happen only with negligible probability because of the collision resistance of $H$. If a real-world honest verifier accepts to update to a functionality where the signer is honest, it must be because she received the update information properly signer with respect to the cold-key of the honest signer. We recall that any time that an honest signer sends this update information, our simulator also requests $\mathsf{U}^{\mathscr{F}_{\mathtt{SIG}}}_{\mathtt{SttUp}_{\mathtt{SIG}},\mathtt{UpPred}_{\mathtt{SIG}}}$ to create a new node in the graph (this is a consequence of the honest signer receiving the command UPDATE). Hence, if case 3) happens it must be that the honest signer did not receive an update command which yielded to the creation of a functionality with sub-session-id ssid , but the verifier received a valid signature for update information that yielded an honest verifier to update to a functionality with sub-session-id ssid. Given that we have excluded that any of the previous case can happen, and given that the PKI provides the correct cold-key for any given signer, the only reason why 3) can occur is because the cold-signature scheme has been forged. But by assumption, this can happen only with negligible probability. □

### 4.3 Practical Impact: Preparation for Post-Quantum Readiness

Our construction in Section 4.2 is written in a formal language that abstracts a few key components. First, $\mathcal{F}_{\mathsf{Comp}}$ models the belief about which parts of the system might fail by reasoning about two types of signatures: hot and cold; second, the construction is based on signature functionalities and thus fully generic, and third, our solution just requires a very simple root of trust (CA without

updates). It is important to map these abstractions to real-world use cases to appreciate the value of the protocol.

Assume you had a signing service that currently cannot afford to sign with anything less efficient than (DL-based) systems like ECDSA or EdDSA. At the same time, you would like to be ready to transition to a signature scheme that is based, say, on lattices as soon as either (i) the quantum threat (against authentication schemes) becomes real or (ii) a post-quantum signature scheme has been developed with efficiency as ECDSA and widely deployed to enable a frictionless operation post-transition.

When you "flip the switch" to the new signature algorithm, your system should definitely not forget the documents it has signed in the past. Clearly, if all you do is to sign old documents issued earlier with a new secret key, then an attacker can impersonate your service unless you cryptographically connect your new key to your old key to preserve the link. This latter step requires interaction, either through a (post-quantum ready) PKI that allows you to record your key update, or via a direct method (as in our solution) secured by an additional scheme that remains secure despite the threat (the cold signature scheme).

Therefore, when used to prepare for a transition in the context of post-quantum readiness, our scheme has the following features.

- When the initial public key is distributed (before the quantum threat is practical), our construction does not rely on a quantum secure PKI since we do not have to update our key in the PKI, but we secure the update ourselves based on this initial root of trust (e.g., downloaded by parties or stored in the browser). This is an important feature for another reason: the developer of a post-quantum ready PKI for example cannot rely itself on a post-quantum secure PKI as this would end up in a circular dependency.

- The update itself is secured by a post-quantum secure signature (corresponding in the formal model to what we call cold signature scheme). Since we do not use it often, this can be a signature scheme that is not very efficient or even limited, and therefore be based on extremely conservative assumptions like one-way functions.

- The system we transition to can be some new algorithm that was not known at the time we set up the initial public key. We can transition to the most favorite signing algorithm at the point of the update. This means that the decision of which post-quantum signature to use in the daily business can be deferred to the point when the need arises (where presumably a better understanding of post-quantum signatures will facilitate more efficient schemes).

Our protocol gives a way to be post-quantum ready w.r.t. signatures without relying on external post-quantum secure services to be ready, and without committing *now* to the signature scheme used after the transition.

### 4.4 A Succinct Representation of the Update System

It is interesting to see how a "compiled" version of $\mathsf{U}^{\mathscr{F}_{\mathtt{SIG}}}_{\mathtt{SttUp}_{\mathtt{SIG}},\mathtt{UpPred}_{\mathtt{SIG}}}$ formally compares to the standard signature functionality. While intuitively clear based on the above statements, we can make the claim more formal by introducing the following protocol $\pi^{S}_{upd}$ that wraps the update system in the most obvious way to obtain an abstraction that then looks like a signature functionality with updates:

1. All machines keep track of a local counter $ep$ of the number of successful updates, initially $ep \leftarrow 0$.

2. On input $(\textsc{key-gen}, \mathrm{sid})$ or $(\textsc{sign}, \mathrm{sid}, m)$ the signer $S$ issues this command to $\mathsf{U}^{\mathscr{F}_{\mathtt{SIG}}}_{\mathtt{SttUp}_{\mathtt{SIG}},\mathtt{UpPred}_{\mathtt{SIG}}}$ and returns the result to the caller.

3. On $(\textsc{verify}, \mathrm{sid}, m, \sigma, vk')$ relay the verification request to $\mathsf{U}^{\mathscr{F}_{\mathtt{SIG}}}_{\mathtt{SttUp}_{\mathtt{SIG}},\mathtt{UpPred}_{\mathtt{SIG}}}$ and return any answer to the caller.

4. On $(\textsc{update}, \mathrm{sid})$ for signer $S$, issue the update request $(\textsc{update}, \mathrm{sid}, \widehat{\mathcal{F}}^{S,\mathcal{DS}_{ep+1}}_{\mathtt{SIG}})$ to $\mathsf{U}^{\mathscr{F}_{\mathtt{SIG}}}_{\mathtt{SttUp}_{\mathtt{SIG}},\mathtt{UpPred}_{\mathtt{SIG}}}$ and return any answer to the caller. Upon a successful update notification $(\mathsf{Success}, \mathrm{ssid}, F)$ set $ep \leftarrow ep + 1$. If the update fails, abort the execution.

5. On $(\textsc{update}, \mathrm{sid})$, issue the update request $(\textsc{update}, \mathrm{sid}, \widehat{\mathcal{F}}^{S,\mathcal{DS}_{ep+1}}_{\mathtt{SIG}})$ to $\mathsf{U}^{\mathscr{F}_{\mathtt{SIG}}}_{\mathtt{SttUp}_{\mathtt{SIG}},\mathtt{UpPred}_{\mathtt{SIG}}}$. Upon a successful update notification $(\mathsf{Success}, \mathrm{ssid}', \widehat{\mathcal{F}}^{S,\mathcal{DS}_{ep+1}}_{\mathtt{SIG}})$ set $ep \leftarrow ep + 1$. If the update fails or the identities do not match, abort the execution.

In Figure 7 we describe the concise signature functionality with updates that the above protocol realizes, and which has the following features: we have consistency throughout the execution of the system as long as the signer is not fully corrupted. This means that no party will see a tuple $(m, \sigma, vk)$ fail to verify if it did successfully verify in the past. Next, if the signer updates before it is sub-corrupted, the set of signed messages remains exactly the set of messages formed by the messages that were input by the signer. In this case, every verifier in the same epoch as the signer enjoys the normal unforgeability guarantees—and always with respect to the one registered public key of the signer—despite the fact that forgeries are allowed for older epochs or during periods of signer sub corruptions. We can summarize this observation as a lemma.

**Lemma 4.3.** *Protocol $\pi^S_{upd}$ defined above UC-realizes $\mathcal{F}^{\star,S}_{\mathtt{SIG}}$.*

*Proof.* We first describe the simulator $\mathcal{S}$ for this construction that interacts with $\mathcal{F}^{\star,S}_{\mathtt{SIG}}$. The simulator internally emulates the instance of $\mathsf{U}^{\mathscr{F}_{\mathtt{SIG}}}_{\mathtt{SttUp}_{\mathtt{SIG}},\mathtt{UpPred}_{\mathtt{SIG}}}$. It behaves as follows from inputs from $\mathcal{F}^{\star,S}_{\mathtt{SIG}}$:

- On receiving $(\textsc{key-gen}, \mathrm{sid})$, $(\textsc{sign}, \mathrm{sid}, m)$, or $(\textsc{verify}, \mathrm{sid}, m, \sigma, v')$ from a particular party $P$, hand the corresponding input to the emulated instance $ep_P$ of the session $\mathsf{U}^{\mathscr{F}_{\mathtt{SIG}}}_{\mathtt{SttUp}_{\mathtt{SIG}},\mathtt{UpPred}_{\mathtt{SIG}}}$ and send to the environment the output that is produced. Return the answer from the environment (which is either $(\textsc{verification-key}, \mathrm{sid}, v)$, $(\textsc{signature}, \mathrm{sid}, m, \sigma)$, or $(\textsc{verified}, \mathrm{sid}, m, \phi)$) to any of these requests back to the very same instance and forward the reply to $\mathcal{F}^{\star,S}_{\mathtt{SIG}}$ to produce the matching output towards $P$.

- On receiving $(\textsc{SignerAdvances}, \mathrm{sid}, ep_S)$ the simulator $\mathcal{S}$ simulates the update request $(\textsc{update_notification}, \mathrm{sid}, \mathsf{UpdateReq}_F, S, \widehat{\mathcal{F}}^{S,\mathcal{DS}_{ep_S+1}}_{\mathtt{SIG}})$ (where each $\mathsf{UpdateReq}_F$ is simply filled with all parties that want to update to $F$) to the environment.

- On receiving $(\textsc{Update}, \mathrm{sid}, P)$, the simulator $\mathcal{S}$ simulates the update request analogous to above, i.e., by sending $(\textsc{update_notification}, \mathrm{sid}, \mathsf{UpdateReq}_F, P, \widehat{\mathcal{F}}^{S,\mathcal{DS}_{ep_P+1}}_{\mathtt{SIG}})$ to the environment.

- On receiving $(\textsc{GrantUpdate}, \mathrm{sid}, P, \mathrm{ssid}, F', aux)$, the simulator internally evaluates the request on its running instance. If the update is successful, it returns $(\textsc{AllowUpdate}, \mathrm{sid}, 1)$ to $\mathcal{F}^{\star,S}_{\mathtt{SIG}}$, and if the update is not successful, it returns $(\textsc{AllowUpdate}, \mathrm{sid}, 1)$ to $\mathcal{F}^{\star,S}_{\mathtt{SIG}}$.

This simulation yields an execution that is indistinguishable from an execution of $\pi_{up}$. This is obvious before the first update, as both worlds simply talk to an ideal signature functionality they have an identical behavior by definition. Furthermore, if the signer ever gets sub-corrupted, both

worlds weaken the ideal unforgeability guarantees and the simulator is free to decide on the outputs by $S$. Regarding the updates, as long as the signer is honest, the update, i.e., epoch changes can only be triggered by the signer by definition of $\texttt{UpPred}_{\texttt{SIG}}$, as otherwise, no child node exists. Furthermore, the update graph is in fact a chain: the only admissible ssid for everyone is the ssid obtained by the signer $S$ when the update succeeds and the functionality must be the one requested by the signer, because an honest signer only performs one update a time and $\texttt{UpPred}_{\texttt{SIG}}$ would only let this one update go through and otherwise the update fails. Thus, as long as the signer is honest, both systems behave identically. We are thus left with analyzing the case that the signer becomes fully corrupted. To this end, let $ep^*$ denote the epoch during which the signer becomes fully corrupted. We observe that the ideal system lifts the unforgeability and global consistency guarantees and the simulator is free to decide the output of a signature verification. The only restriction that is maintained is local consistency. However, by definition of $\texttt{UpPred}_{\texttt{SIG}}$, no matter what transition a party $P$ does, local consistency is retained also in the real world. This concludes the proof. □

# 5 Non-interactive Zero-Knowledge Proof Systems

We define the update system for non-interactive zero-knowledge proof systems. The update protocol follows a similar pattern, but is privacy-preserving admitting no more leakage about the witnesses beyond what is leaked due to corruptions.

## 5.1 The NIZK Functionality

**NIZK with subroutine corruption.** We define the NIZK functionality from [GOS12] in our framework, which simply means that, as in the previous section, we formally equip it with the standard sub-routine corruption mode as outlined in Section 3.1 and Section A and refer to Section A.2.1 for a description of $\mathcal{F}_{\texttt{NIZK}}$. Recall that a proof system is called a UC-NIZK scheme if, cast as a straightforward UC protocol $\Pi_{\mathcal{PS}}$, UC-realizes $\mathcal{F}_{\texttt{NIZK}}$ (for simplicity, we focus on the use-case with one specific prover). For full generality of our modular update system, we formalize several forms of CRS corruptions (as our security statements do hold w.r.t. any such form of corruption), by formally defining a CRS functionality in Figure 13 in Section A that outlines several assumptions about the corruptibility of the CRS. Note that we only capture leakage about the CRS, as subversion is captured by our formalism using the computation devices $\mathcal{F}_{\mathsf{Comp}}$.

Analogously to the previous section, the subroutine-corruptible version is denoted by $\widehat{\mathcal{F}}_{\texttt{NIZK}}$, which is UC-realized by running the proof system in the $\mathcal{F}_{\mathsf{Comp}}$-hybrid world (modeling algorithm subversion), and in the $\mathcal{F}_{\mathsf{CRS}}$-hybrid world (where we drop the reference to the scheme because it is not relevant for our considerations below) to model CRS-corruption. The protocol is called $\widehat{\Pi}_{\mathcal{PS}}$ and we have the following lemma analogous to the previous section.

**Lemma 5.1.** *If $\mathcal{PS}$ is a proof system such that $\Pi^{\mathtt{P}}_{\mathcal{PS}}$ UC-realizes $\mathcal{F}^{\mathtt{P}}_{\texttt{NIZK}}$ in the $\mathcal{F}_{\mathsf{CRS}}$-hybrid model then $\widehat{\Pi}^{\mathtt{P}}_{\mathcal{PS}}$ realizes $\widehat{\mathcal{F}}_{\texttt{NIZK}}$ in the $(\mathcal{F}_{\mathsf{Comp}}, \mathcal{F}_{\mathsf{CRS}})$-hybrid model.*

*Proof.* The only difference between $\widehat{\Pi}^{\mathtt{P}}_{\mathcal{PS}}$ and $\Pi^{\mathtt{P}}_{\mathcal{PS}}$ is that the algorithms $\mathsf{Prove}$ and $\mathsf{Vrfy}$ are executed on $\mathcal{F}_{\mathsf{Comp}}$ by the former. If no subroutine-corruptions on $\mathcal{F}_{\mathsf{Comp}}$ occur, then the simulator $\mathcal{S}$, which exists by assumption (simulating w.l.o.g. against the dummy real-world adversary) provides an indistinguishable ideal execution w.r.t. $\widehat{\Pi}^{\mathtt{P}}_{\mathcal{PS}}$ in the same corruption model. In case of subroutine corruptions of $\mathcal{F}_{\mathsf{Comp}}$ for prover $\mathtt{P}$, the behavior in both worlds is equivalent to a full corruption of $\mathtt{P}$ which can be simulated by instructing $\mathcal{S}$ to corrupt $\mathtt{P}$ and providing back the information obtained

**Functionality $\mathcal{F}_{\mathtt{SIG}}^{\star,S}$**

The functionality interacts with parties $P \in \mathcal{P}$ and initializes $ep_P \leftarrow 0$, $\mathcal{M}_P \leftarrow \emptyset$ for all $P \in \mathcal{P}$, as well as a set $\mathcal{M} \leftarrow \emptyset$. The function is parameterized by the signer identity $S \in \mathcal{P}$. The corruption set consists of identities $P \in \mathcal{P}$, (full corruption) or $(\mathcal{F}_{\mathsf{Comp}}, s, P)$ for $P \in \mathcal{P}$ (subroutine corruption).

- Upon receiving $(\textsc{key-gen}, \mathrm{sid})$ from signer $S$ for the first time, hand $(\textsc{key-gen}, \mathrm{sid})$ to the $\mathcal{A}$. Upon receiving $(\textsc{verification-key}, \mathrm{sid}, vk)$ from $\mathcal{A}$, verify that no entry $(\cdot, m, \cdot, vk, 1)$ is recorded and ignore the reply if there is such an entry. Else, record the pair $(S, vk)$ and output $(\textsc{verification-key}, \mathrm{sid}, vk)$ to $S$. Otherwise, ignore the request.
- Upon receiving $(\textsc{sign}, \mathrm{sid}, m)$ is received from signer $S$, send $(\textsc{sign}, \mathrm{sid}, m)$ to $\mathcal{A}$. Upon receiving $I = (\textsc{signature}, \mathrm{sid}, m, \sigma)$ from $\mathcal{A}$, verify that no entry $(ep, m, \sigma, vk, 0)$ for any $ep$ is stored. If it is, then ignore the reply. Else, send $(\textsc{signature}, \mathrm{sid}, m, \sigma)$ to $S$, and store the record $(ep_S, m, \sigma, vk, 1)$ in $\mathcal{M}$. Additionally, store the record $(m, \sigma, vk, 1)$ in $\mathcal{M}_S$.
- Upon receiving $(\textsc{verify}, \mathrm{sid}, m, \sigma, vk')$ from some party $P$, hand $(\textsc{verify}, \mathrm{sid}, m, \sigma, vk')$ to the adversary. Upon receiving $(\textsc{verified}, \mathrm{sid}, m, \phi)$ from the adversary do:
  1. If $vk' = vk$ and $S \notin \mathcal{C}$, and the entry $(ep_P, m, \sigma, vk, 1) \in \mathcal{M}$ is recorded, then set $f \leftarrow 1$. *(Completeness)*
  2. Else, if $vk' = vk$ and $S \notin \mathcal{C}$ and $(\cdot, ep_P, S) \notin \mathcal{C}$, and if no entry $(ep_P, m, \cdot, vk, 1)$ is already stored, then set $f \leftarrow 0$ and record the entry $(ep_P, m, \sigma, vk, 0)$. *(Unforgeability)*
  3. Else, if the entry $(ep_P, m, \sigma, vk', f') \in \mathcal{M}$ is recorded and $S \notin \mathcal{C}$ then let $f \leftarrow f'$. *(Consistency across all parties in epoch)*
  4. Else, if the entry $(m, \sigma, vk', f') \in \mathcal{M}_P$ is recorded, then let $f \leftarrow f'$. *(Local consistency)*
  5. Else, let $f \leftarrow \phi$. If $S \notin \mathcal{C}$, record the entry $(ep_P, m, \sigma, vk', \phi)$.

  Store the tuple $(m, \sigma, vk', f)$ in $\mathcal{M}_P$ if the result is $f = 1$. Output $(\textsc{verified}, \mathrm{sid}, m, f)$ to $P$.
- Upon receiving $(\textsc{Update}, \mathrm{sid})$ from signer $S$, output $(\textsc{SignerAdvances}, \mathrm{sid}, ep_S)$ to the adversary. Upon receiving $(\textsc{AllowUpdate}, \mathrm{sid}, d)$ from the adversary then do the following: if $d = 0$ then abort; otherwise set $ep_S \leftarrow ep_S + 1$ and store for each record $(m, \sigma, vk, b) \in \mathcal{M}_S$ the record $(ep_S, m, \sigma, vk, b)$.
- Upon receiving $(\textsc{Update}, \mathrm{sid})$ from some party $P \in \mathcal{P}$ perform the following: hand $(\textsc{Update}, \mathrm{sid}, P)$ to the adversary. Upon receiving the answer $(\textsc{AllowUpdate}, \mathrm{sid}, d)$, do the following: if $d = 0$, then abort. If $d = 1$ and if $ep_P < ep_S$ then set $ep_P \leftarrow ep_P + 1$ and store for each record $(m, \sigma, vk, b) \in \mathcal{M}_P$ the record $(ep_P, m, \sigma, vk, b)$ in $\mathcal{M}$.
- The functionality follows the fine-grained corruption mode explained in Section 3.1.

Figure 7: The signature functionality $\mathcal{F}_{\mathtt{SIG}}^{\star,S}$ with explicit updates.

from $\widehat{\mathcal{F}}_{\texttt{NIZK}}$ when sub-corrupting $\texttt{P}$. From then on, simulation is trivial, as no private information is retained, and we have complete control over the output of $\texttt{P}$. For any other party, i.e., verifier, simulation is trivial given the equivalence of full party corruption and verifier sub-corruption. $\square$

## 5.2 Ideal Updatable NIZK Proof Systems

The ideal update system that we achieve is fully specified by defining the core functions and noting that we aim to realize $\mathsf{U}^{\mathscr{F}_{\texttt{NIZK}}}_{\texttt{SttUp}_{\texttt{NIZK}},\texttt{UpPred}_{\texttt{NIZK}}}$, for $\mathscr{F}_{\texttt{NIZK}} := \{\widehat{\mathcal{F}}^{\texttt{P},\mathcal{PS}_i}_{\texttt{NIZK}}\}_i$, the family of NIZK functionalities as defined above which are formally indexed with $\mathcal{PS}_i$. Note that $\texttt{P}$ appears as explicit identifier in this family.

- $\texttt{UpPred}_{\texttt{NIZK}}$: On input $(P, \text{ssid}, \mathcal{F}', \mathsf{UpdateReq}_{\mathcal{F}'}, \mathsf{UpdateStatus}_{P,\mathcal{F}'}, \mathsf{PartiesFunctions}, \mathsf{UpGraph}, 0^\lambda)$ do the following: if $P$ appears in $\mathsf{UpdateReq}$ and $\mathsf{UpdateStatus}_{P,\mathcal{F}'} = \textit{Updating}$ then let $v \leftarrow \mathsf{PartiesFunctions}[P]$ and perform the following steps to compute the decision (otherwise, return $\perp$):
  - If $P$ is the prover of instance $v$ and $\mathsf{UpGraph}$ has no node $v'$ with $v'.\mathsf{function} = \mathcal{F}'$ and $v'.\text{ssid} = \text{ssid}$ then return 1.
  - Else, if P is not a prover and there is a node $v'$ with $v'.\mathsf{function} = \mathcal{F}'$ and $v'.\text{ssid} = \text{ssid}$ that is a child of $v$ then do the following.
    - If $\texttt{P}$ is honest then return 1.
    - If $\texttt{P}$ is corrupted then let $M_P$ be the set of all the pairs $(x, \pi)$ s.t. $(P, x, w, \pi) \in v.\mathsf{state}$ and let $M$ be the all the pairs $(x, \pi)$ for which a tuple $(\cdot, x, w, \pi)$ is stored in $v'.\mathsf{state}$, if $M_P \subseteq M$ then return 1.
  - Return 0.

This is analogous to the previous section: we want to upgrade to a more secure system while maintaining backwards-compatibility.

- $\texttt{SttUp}_{\texttt{NIZK}}$: On input $\texttt{SttUp}(v^\star, P, \mathsf{UpdateReq}, \mathsf{PartiesFunctions}, \mathsf{UpGraph}, \text{aux})$ first compute $v \leftarrow \mathsf{PartiesFunctions}[P]$ (note that by definition, $v^\star$ is the parent node of $v$ party $P$ is updating from).
  - If $P$ is the prover in instance $v^\star$ and $P$ is honest, define the new state $s$ and insert all tuples $(\texttt{P}, x, w, \pi) \in v^\star.\mathsf{state}$.
  - Else, if $P$ is the prover for the instance $v^\star$ and $P$ is corrupted then define an empty state $s$, and set $s \leftarrow aux$.
  - If $P$ is not the prover then set $s \leftarrow v.\mathsf{state}$.
  - Return $s$.

The state update is again similar to the previous update system: all we can guarantee is that if the prover is honest, all honest parties will upgrade to the next instance and regain all guarantees from that point onwards. However, if the prover is dishonest, it can decide about the new state, potentially preventing other parties to proceed with the update.

**Protocol $\Pi^{\mathtt{P},up}_{\mathtt{NIZK}}$ - Part I**

*Initialization*

Let $\mathcal{DS}_c = (\mathsf{KGen}_c, \mathsf{Sign}_c, \mathsf{Ver}_c)$ be the *cold* signature scheme. Each party $P \in \mathcal{P}$ initializes $\mathtt{Version}^P \leftarrow 0$, $\mathrm{sid}^P \leftarrow \mathrm{sid}$ (where sid is the session id of the protocol instance), maintains an intially empy set $M^P$ and defines $\mathsf{Vrfy}^P$ which on input $(x, \pi)$ sends $(\textsc{verify}, \mathrm{sid}, x, \pi)$ to $\widehat{\mathcal{F}}^{\mathtt{P},\mathcal{PS}_0}_{\mathtt{NIZK}}$ and returns whatever is returned by the functionality.

The prover $\mathtt{P}$ maintains also a set denoted with $W$ (initially empty) and does the following additional steps:

- Generate $(vk_c, sk_c) \leftarrow \mathsf{KGen}_c(1^\lambda)$ and register $vk_c$ with the PKI.
- Define $\mathsf{Prove}^{\mathtt{P}}$ which is a stateful algorithm that on input statement $x$ and a witness $w_x$ performs the following steps:
  - Send $(\textsc{prove}, \mathrm{sid}, x, w_x)$ to $\widehat{\mathcal{F}}^{\mathtt{P},\mathcal{PS}_0}_{\mathtt{NIZK}}$.
  - Upon receiving $\pi$ from the functionality, add $(x, \pi)$ to $M^{\mathtt{P}}$ and $(x, w_x)$ to $W$ and return $(x, \pi)$.

*Proving and Verifying:*

- Upon receiving the command $I := (\textsc{input}, \mathrm{sid}, y)$,
  - if $y = (\textsc{prove}, x, w_x)$ and $I$ is received by $\mathtt{P}$, then compute $\mathsf{Prove}^{\mathtt{P}}(x, w_x)$ thus obtaining $(x, \pi)$ and return $(\textsc{proof}, \mathrm{sid}, x, \pi)$;
  - if $y = (\textsc{verify}, x, \pi)$ and $I$ is received by a party $P$ that is not the prover then check whether $\mathsf{Vrfy}^P(x, \pi) = 1$. If this is the case, do the following:
    - add $(x, \pi)$ to $M^P$;
    - return 1.

    else return 0.

Figure 8: Updatable Proof Systems - Initial Operation

## 5.3 Protocol and Security Statement

The protocol is defined in Figures 8 and 9. It follows the same basic idea as in the previous section, i.e., the prover needs to sign update information that is sufficient to safely move to the next instance of $\widehat{\mathcal{F}}_{\mathtt{NIZK}}$ while still being backwards compatible. Nevertheless, there is a catch: when a malicious prover is initiating an update, it should not be possible to "inject" statements $x$ which are not in the language. Recall that since a verifier's algorithm might be subverted at the time of the update, and since we want to implement backwards-compatibility as specified above, we must rely on a cold-scheme mechanism to detect invalid statements if we want to retain the NIZK security guarantee for the next epoch (we note that dropping the backwards compatibility requirement is always possible and we end up with a simple problem as seen in Section C). The cold scheme mechanism is used in a batch-verification step that happens during the update. We obtain:

**Theorem 5.2.** *Let each proof system $\mathcal{PS}_i$ be a UC-NIZK systems for NP relation $R$ (with respect to some corruption mode) in the $(\mathcal{F}^{\mathsf{Gen}_i}_{\mathsf{CRS}}, \mathcal{F}_{\mathsf{Comp}})$ model, then there is a NIZK update protocol (w.r.t. the same corruption mode), that is, there is a protocol realizing $\mathsf{U}^{\mathscr{F}_{\mathtt{NIZK}}}_{\mathtt{SttUp}_{\mathtt{NIZK}},\mathtt{UpPred}_{\mathtt{NIZK}}}$ for $\mathscr{F}_{\mathtt{NIZK}}$ based on a collision-resistant hash function, the availability of a (cold) signature scheme and a PKI, as well as a cold UC-NIZK scheme for the batch-proof relation $R' = \{(x_1, \ldots, x_n), (w_1, \ldots, w_n) \mid n \in \mathbb{N} \wedge (x_i, w_i) \in R\}$ which is only used during the update step.*

**Protocol $\Pi^{\mathtt{P},up}_{\mathtt{NIZK}}$ - Part II**

*Update:*

- If the prover $\mathtt{P}$ receives the command $I := (\textsc{update}, \mathrm{sid}, \widehat{\mathcal{F}}^{\mathtt{P},\mathcal{PS}_\star}_{\mathtt{NIZK}})$ with $\widehat{\mathcal{F}}^{\mathtt{P},\mathcal{PS}_\star}_{\mathtt{NIZK}} \in \mathscr{F}_{\mathtt{NIZK}}$ then $\mathtt{P}$ sets $\mathrm{sid}^{\mathsf{P}} \leftarrow H(M^{\mathsf{P}}||\widehat{\mathcal{F}}^{\mathtt{P},\mathcal{PS}_\star}_{\mathtt{NIZK}})$ does the following steps.

  1: Identify the elements of $W$ and $M^{\mathsf{P}}$ as the sequences $S_W = (x_i, w_i)_{i\in[|W|]}$ and $S_M = (x_i, \pi_i)_{i\in[|M^{\mathsf{P}}|]}$, respectively, such that $S_W[i,1] = S_M[i,1]$.

  2: Build the vectors $x = (x_1, \ldots, x_{|W|})$ and $w = (w_1, \ldots, w_{|W|})$ and send $(\textsc{prove}, \mathrm{sid}, x, w)$ to $\mathcal{F}^{\mathtt{P},\mathcal{PS}_c}_{\mathtt{NIZK}}$, and receive $\pi_{up}$ as a reply.

  3: Redefine $\mathsf{Prove}^{\mathsf{P}}$ as follows: on input $(x, w)$, the algorithm does the following:

  - Send $(\textsc{prove}, \mathrm{sid}^{\mathsf{P}}, x, w_x)$ to $\widehat{\mathcal{F}}^{\mathtt{P},\mathcal{PS}_\star}_{\mathtt{NIZK}}$.
  - Upon receiving $\pi$ as a reply, add $(x, w_x)$ to $W$ and $(x, \pi)$ to $M^{\mathsf{P}}$.

  4: Set $\mathtt{Version} \leftarrow \mathtt{Version} + 1$ and compute $h \leftarrow H(M^{\mathsf{P}}||\widehat{\mathcal{F}}^{\mathtt{P},\mathcal{PS}_\star}_{\mathtt{NIZK}})$.

  5: Compute $\sigma_c \leftarrow \mathsf{Sign}_c(sk_c, \mathtt{Version}||h)$, define $\tau \leftarrow \widehat{\mathcal{F}}^{\mathtt{P},\mathcal{PS}_\star}_{\mathtt{NIZK}}||\mathtt{Version}||M^{\mathsf{P}}||h||\pi_{up}||\sigma_c$ and send $\tau$ to each verifier.

- If a party $P$ (which is not the prover) receives the command $I := (\textsc{update}, \mathrm{sid}, \widehat{\mathcal{F}}^{\mathtt{P},\mathcal{PS}_\star}_{\mathtt{NIZK}})$ with $\widehat{\mathcal{F}}^{\mathtt{P},\mathcal{PS}_\star}_{\mathtt{NIZK}} \in \mathscr{F}_{\mathtt{NIZK}}$, then she does the following steps.

  1: Wait to receive $\tau$ from the signer and parse it as $\mathcal{F}||\mathtt{Version}||M||h||\pi||\sigma_c$, set $\mathrm{sid}^P_c \leftarrow H(M||\widehat{\mathcal{F}}^{\mathtt{P},\mathcal{PS}_\star}_{\mathtt{NIZK}})$ and do the following checks: $\mathsf{Ver}_c(vk_c, \mathtt{Version}||h, \sigma_c) = 1$ (where $vk_c$ of $\mathtt{P}$ is obtained from the PKI) and $h = H(M||\mathcal{F})$ and $\mathcal{F} = \widehat{\mathcal{F}}^{\mathtt{P},\mathcal{PS}_\star}_{\mathtt{NIZK}}$ and $\mathtt{Version} > \mathtt{Version}^P$ and

  - build the sequence $x_M = (x_1, \ldots, x_{|M|})$ from $M$ and send $(\textsc{verify}, \mathrm{sid}, x_M, \pi_{up})$ to $\mathcal{F}^{\mathtt{P},\mathcal{PS}_c}_{\mathtt{NIZK}}$ and upon receiving $b$, check if $b = 1$,
  - check if $M^P \subseteq M$.

  If any of the above checks fails then then ignore the input, else set $M^P \leftarrow M$ and $\mathrm{sid}^P \leftarrow h$ and redefine $\mathsf{Vrfy}^P$ on input $(x, \pi)$ as follows:

  - Send $(\textsc{verify}, \mathrm{sid}^P, x, \pi)$ to $\widehat{\mathcal{F}}^{\mathtt{P},\mathcal{PS}_\star}_{\mathtt{NIZK}}$ and upon receiving $b$, if $b = 1$ then
    - add $(x, \pi)$ to $M^P$
    - return 1.
  - else, if $(x, \pi) \in M^P$ then return 1, else return 0.

Figure 9: Updatable Proof Systems - Update Step

The theorem follows by Lemma 5.3 and by the composition theorem to replace $\widehat{\mathcal{F}}_{\mathtt{NIZK}}$ in $\Pi^{\mathtt{P},up}_{\mathtt{NIZK}}$.

**Lemma 5.3.** *Let $\mathscr{F}_{\mathtt{NIZK}} := \{\widehat{\mathcal{F}}^{\mathtt{P},\mathcal{PS}_i}_{\mathtt{NIZK}}\}$ be a family of functionalities as defined above which are formally indexed with $\mathcal{PS}_i$. Assuming $H$ is a collision resistant hash function, $\mathcal{DS}_c$ is a signature scheme, and the availability of a PKI, then $\Pi^{\mathtt{P},up}_{\mathtt{NIZK}}$ realizes $\mathsf{U}^{\mathscr{F}_{\mathtt{NIZK}}}_{\mathtt{SttUp}_{\mathtt{NIZK}},\mathtt{UpPred}_{\mathtt{NIZK}}}$.*

*Proof sketch.* The proof shares a lot of similarities with the proof of the signature update system. In particular, the simulator $\mathcal{S}$ interacts with $\mathsf{U}^{\mathscr{F}_{\mathtt{NIZK}}}_{\mathtt{SttUp}_{\mathtt{NIZK}},\mathtt{UpPred}_{\mathtt{NIZK}}}$ and must simulate the real-world protocol execution (w.r.t. the dummy real-world adversary). Since we work in the hybrid world $\widehat{\mathcal{F}}_{\mathtt{NIZK}}$ the interaction before the update is straightforward to simulate. During an update, we can distinguish the two cases whether the prover $\mathtt{P}$ is honest (and possibly sub-corrupted) or corrupted.

- If the prover is honest, $\mathcal{S}$ computes the update information $\tau = \widehat{\mathcal{F}}^{\mathtt{P},\mathcal{PS}_\star}_{\mathtt{NIZK}}||\mathtt{Version}||M^{\mathsf{P}}||h||\pi_{up}||\sigma_c$,

where $M^{\mathtt{P}}$ is obtained by the emulation of hybrid $\widehat{\mathcal{F}}_{\mathtt{NIZK}}$ (where the proofs are given input by the dummy adversary per instruction by the environment). Similarly, $\pi_{up}$ is the proof obtained when the simulator simulates the output $(\textsc{prove}, \mathrm{sid}, (x_i, \ldots, x_{|M^{\mathtt{P}}|}))$ to the environment and grants the update to every party that receives this information with ssid equal to $h$. This simulation works by the same arguments as in the previous section. As long as the cold signature scheme is not broken and no hash-function collision occurs, there can only be one instance of the updated proof system where the initial state consists of all prior proofs to statements proven by $\mathtt{P}$.

- If the prover is dishonest then each update information $\tau_i = \widehat{\mathcal{F}}^{\mathtt{P},\mathcal{PS}_\star}_{\mathtt{NIZK}} || \mathtt{Version} || M^{\mathtt{P}} || h || \pi_{up} || \sigma_c$ sent by the adversary to an honest verifier $P_i$ can be interpreted by $\mathcal{S}$ as spawning an instance of $\widehat{\mathcal{F}}_{\mathtt{NIZK}}$ in session ssid $= h = H(M_i || \widehat{\mathcal{F}}^{\mathtt{P},\mathcal{PS}_\star}_{\mathtt{NIZK}})$, where $M_i$ defines the preserved state of the functionality and $\mathcal{PS}_i$ matches the update request by the honest (and potentially sub-corrupted) verifier, i.e., it has received $(\textsc{update}, \mathrm{sid}, \widehat{\mathcal{F}}^{\mathtt{P},\mathcal{PS}_i}_{\mathtt{NIZK}})$. The simulator thus inspects whether $\pi_{up}$ is valid and if it is, obtains the witnesses $(w_1, \ldots, w_{|M_i|})$. Since $\mathtt{P}$ is corrupted at this point by assumption, all witnesses, by definition of $\mathcal{F}_{\mathtt{NIZK}}$ are either known by corrupting $\mathtt{P}$ in all instances of $\widehat{\mathcal{F}}_{\mathtt{NIZK}}$ of $\mathsf{U}^{\mathscr{F}_{\mathtt{NIZK}}}_{\mathtt{SttUp}_{\mathtt{NIZK}},\mathtt{UpPred}_{\mathtt{NIZK}}}$ or latest at the time when $\mathcal{S}$ outputs $(\textsc{verify}, \mathrm{sid}, P, x, \pi_{up})$ to the environment (in case $\pi_{up}$ has never been generated before), upon which the witnesses $(w_1, \ldots, w_{|M_i|})$ must be presented. If all simulated checks succeed for the honest verifier, the simulator ensures (by proving a respective statement using the corresponding witness in the name of the corrupted prover) that all tuples $(\mathtt{P}, x_k, w_k, \pi)$ presented in $M_i$ are indeed in the state of the session of $\widehat{\mathcal{F}}_{\mathtt{NIZK}}$ to which $P_i$ is currently connected. Subsequently, $\mathcal{S}$ grants the update for $P_i$ with ssid $= h$. This simulation yields an indistinguishable behavior from the real world as long as no hash collision occurs, because each ssid refers to a unique set of pairs $(x_k, \pi_k)$ proven to be in $R$ during the update, and a honest verifier only proceeds with the update if its own set of previously successfully verified statement and proofs is a subset thereof. Thus, any two verifiers with matching ssid will have the same verification behavior on the the same set of old statements and new verifications (proofs generated anew in session ssid) are verified in exactly the same way in both worlds as both are instances of $\widehat{\mathcal{F}}_{\mathtt{NIZK}}$.

□

# 6 Comparison with a Prior Notion of Agility

We start this section by recalling the notion of crypto-agility introduced in [ABBC10] for the case of pseudo-random functions (the other case treated in [ABBC10], authenticated encryption, could be represented analogously in our framework). Recall Section 2 and Section A for the standard definitions.

**Definition 6.1** (Compatible PRF schemes)**.** We say that a set $\mathcal{S}$ of PRF schemes is compatible if all schemes $\mathsf{prf} \in \mathcal{S}$ have the same key generator.

Consider the games of Figure 10. In [ABBC10] an adversary is called $\mathcal{S}$-restricted if (1) it specifies in its queries only schemes from $\mathcal{S}$, (2) it makes only one **KeySetup** query which is its first oracle query in any execution, (3) it never repeats an oracle query, and (4) any $\mathbf{Fn}(\mathsf{prf}, x)$ query it makes satisfies $x \in \mathcal{X}_\kappa$. (All this must hold with probability 1 regardless of how queries are answered). $\mathcal{S}$ being compatible means the parameter and key generation algorithms invoked during setup will be the same regardless of the PRF scheme that is provided as input to the function evaluation oracle.)

**Game** $\mathsf{FF.PR.Gm}^b$

procedure **KeySetup**($\mathsf{prf}$)
- $K \xleftarrow{\$} \mathsf{prf.KGen}(1^\kappa)$.
- Return $K$

procedure **Fn**($\mathsf{prf}, x$)
- if $b = 1$ then $y \leftarrow \mathsf{prf.Eval}(K, x)$ else $y \xleftarrow{\$} \mathcal{Y}_\kappa$.
- Return $y$.

Figure 10: Game $\mathsf{FF.PR.Gm^b}$ from [ABBC10]. We simplified the description for clarity. We omit the parameters because they are not relevant for our treatment. We formulate security as an equivalent distinguishing problem, and not as a bit-guessing problem.

**Definition 6.2** (PRF agility)**.** Let $\mathcal{A}^{\mathsf{Gm}^b}$ be the output of the adversary in the game $\mathsf{FF.PR.Gm^b}$ of Fig. 10. We say that a finite, compatible set $\mathcal{S}$ of PRF schemes is agile if $|\Pr[\mathcal{A}^{\mathsf{Gm}^0} = 1] - \Pr[\mathcal{A}^{\mathsf{Gm}^1} = 1]|$ is negligible for all efficient $\mathcal{S}$-restricted adversaries.

## 6.1 Comparison with Crypto-Agility from [ABBC10]

In this section, we define a simple update system that leads to the above prior notion of agility. The instantiation is simple since in [ABBC10] the algorithms are always executed honestly and as such, no guarantees in the presence of corruptions are given. We can therefore just consider a single party under the UC static corruption model—we point out that as we have seen in prior sections, our formalism can capture a much richer corruption model (see for example [JT20] on how to obtain PRFs with adaptive security in the random oracle model).

### 6.1.1 Ideal Updatable (Pseudo-) Random Functions

Again, the ideal update system that we achieve is fully specified by defining the core functions and noting that we aim to realize $\mathsf{U}^{\mathscr{F}_{\mathtt{URF}}}_{\mathtt{SttUp}_{\mathtt{URF}},\mathtt{UpPred}_{\mathtt{URF}}}$, for $\mathscr{F}_{\mathtt{URF}} := \{\mathcal{F}^{\mathsf{P},(\mathsf{KGen},\mathsf{Eval}_i)}_{\mathtt{URF},\mathcal{X},\mathcal{Y}}\}_i$, the family of URF functionalities and formally indexed with $\mathsf{prf}_i$, where as explained above, all schemes share the key generation process and the single party in the system is denoted by $\mathtt{P}$.

- $\mathtt{UpPred}_{\mathtt{URF}}$: On input $(\mathtt{P}, \mathrm{ssid}, \mathcal{F}', \mathsf{UpdateReq}_{\mathcal{F}'}, \mathsf{UpdateStatus}_{P,\mathcal{F}'}, \mathsf{PartiesFunctions}, \mathsf{UpGraph})$ do the following: if $\mathtt{P}$ appears in $\mathsf{UpdateReq}$ and $\mathsf{UpdateStatus}_{\mathtt{P},\mathcal{F}'} = \mathit{Updating}$ and return 1 (otherwise, return $\perp$).

- $\mathtt{SttUp}_{\mathtt{URF}}$: On input $\mathtt{SttUp}(v^\star, \mathtt{P}, \mathsf{UpdateReq}, \mathsf{PartiesFunctions}, \mathsf{UpGraph}, \mathrm{aux})$, obtain the instance $v \leftarrow \mathsf{PartiesFunctions}[\mathtt{P}]$ (note that by definition, $v^\star$ is the parent node of $v$ party $P$ is updating from):
  - If $\mathtt{P}$ is honest and $\exists v' \in \mathsf{UpGraph} : v'.\mathsf{function} = v.\mathsf{function}$ then initialize a new state $s := v'.\mathsf{state}$.
  - Else, if $\mathtt{P}$ is honest and a table has been initialized in $v^\star.\mathsf{state}$, then initialize a new state $s$ with an empty table $T$ and return $s$.
  - Else return the empty state.

We observe that if we repeat the function from a previous update, then the queries are answered consistently (the update graph is actually a chain for an honest $\mathtt{P}$). Furthermore, if we switch to an entirely new function, then we check whether the initialization event has happened or not.

### 6.1.2 Protocol and Security Statement

The protocol $\Pi^{\mathsf{P},up}_{\mathtt{URF}}$ is specified as follows:

- Upon initialization, party $\mathtt{P}$ initializes the empty key $sk := \perp$ and the algorithm $\mathsf{Eval}^{\star} := \mathsf{Eval}_0$.
- On input $I := (\textsc{input}, \mathrm{sid}, y)$ where $y = (\textsc{Initialize}, \mathrm{sid})$, do the following: if $sk = \perp$, execute $sk \xleftarrow{\$} \mathsf{KGen}$, otherwise ignore the input.
- On input $I := (\textsc{input}, \mathrm{sid}, y)$ where $y = (\textsc{Eval}, \mathrm{sid}, x)$, verify that $sk \neq \perp$ and ignore the request if this does not hold. Otherwise, return $s \xleftarrow{\$} \mathsf{Eval}^{\star}(sk, x)$.
- On input $I := (\textsc{update}, \mathrm{sid}, \mathcal{F}^{\mathsf{P},\mathsf{prf}_j}_{\mathtt{URF},\mathcal{X},\mathcal{Y}})$ with $\mathcal{F}^{\mathsf{P},\mathsf{prf}_j}_{\mathtt{URF},\mathcal{X},\mathcal{Y}} \in \mathscr{F}_{\mathtt{URF}}$ set $\mathsf{Eval}^{*} := \mathsf{Eval}_j$.

**Security statement.** Let $\mathcal{S}$ be the set containing all the compatible schemes, and the corresponding functionality class $\mathscr{F}_{\mathtt{URF}}$ indexed with these schemes. We show that the security of the above update system follows from the requirement of $\mathcal{S}$ being agile in the sense of [ABBC10]. We further show that our simple update system captures the agility notion of [ABBC10].

**Theorem 6.3.** *Let $\mathscr{F}_{\mathtt{URF}}$ and $\mathcal{S}$ be as defined above. The set $\mathcal{S}$ is agile if and only if protocol $\Pi^{\mathsf{P},up}_{\mathtt{URF}}$ as defined above realizes $\mathsf{U}^{\mathscr{F}_{\mathtt{URF}}}_{\mathtt{SttUp}_{\mathtt{URF}},\mathtt{UpPred}_{\mathtt{URF}}}$ (with respect to static corruption of party $\mathtt{P}$).*

*Proof.* We start the proof by showing that if $\Pi^{\mathsf{P},up}_{\mathtt{URF}}$ as defined above realizes $\mathsf{U}^{\mathscr{F}_{\mathtt{URF}}}_{\mathtt{SttUp}_{\mathtt{URF}},\mathtt{UpPred}_{\mathtt{URF}}}$ then it is also agile for $\mathcal{S}$. Suppose by contradiction that there exists an adversary $\mathcal{A}^{\mathsf{agile}}$ that contradicts Definition 6.2, then we can construct an environment that distinguishes ideal from real world. Such an environment works as follows.

- Upon receiving an oracle query KeySetup($\mathsf{prf}$), from $\mathcal{A}^{\mathsf{agile}}$, initialize the real/ideal world[3] with $\mathsf{prf}$ and provide the input $I := (\textsc{input}, \mathrm{sid}, y)$ with $y = (\textsc{Initialize}, \mathrm{sid})$.
- Upon receiving an oracle query $\mathrm{Fn}(\mathsf{prf}, x)$ from $\mathcal{A}^{\mathsf{agile}}$ then input $I := (\textsc{update}, \mathrm{sid}, \mathcal{F}^{\mathsf{P},\mathsf{prf}}_{\mathtt{URF},\mathcal{X},\mathcal{Y}})$ and if the update is successful then input $I := (\textsc{input}, \mathrm{sid}, y)$ where $y = (\textsc{Eval}, \mathrm{sid}, x)$.
- If an update ever fails, output 0.
- Return whathever $\mathcal{A}^{\mathsf{agile}}$ returns.

From the above description the output of the environment when it interacts with the ideal-functionality (and any simulator) corresponds to the output of $\mathcal{A}^{\mathsf{agile}}$ in $\mathsf{FF.PR.Gm}^0$ unless it observes that an update fails, in which case the output is 0 (an update never fails in the real world). In the complementary case, the output of the environment corresponds to the output of $\mathcal{A}^{\mathsf{agile}}$ in $\mathsf{FF.PR.Gm}^1$ since in the real world, we exactly mimic the game's oracles. The argument holds for all simulators in the given class of functionalities $\mathcal{F}^{\mathsf{P},(\mathsf{KGen},\mathsf{Eval}_i)}_{\mathtt{URF},\mathcal{X},\mathcal{Y}}$, since the simulator does not interact with these functionalities and only in the ideal world updates can fail. Thus, as long as all updates

[3]Formaly, the ideal world functionality is initialized by parametrizeing the root node of the graph with $\mathcal{F}^{\mathsf{P},\mathsf{prf}}_{\mathtt{URF},\mathcal{X},\mathcal{Y}}$, whereas in the real world the parties have access to scheme $\mathsf{prf}$.

are successful, the ideal behaves like $\mathsf{FF.PR.Gm}^0$ since all new function values are sampled uniformly at random from the output domain.

We now want to prove that if $\Pi^{\mathsf{P},up}_{\mathtt{URF}}$ is agile then it also realizes $\mathsf{U}^{\mathscr{F}_{\mathtt{URF}}}_{\mathtt{SttUp}_{\mathtt{URF}},\mathtt{UpPred}_{\mathtt{URF}}}$. A simulator can be specified in a straightforward manner: the simulator always grants an update with what party $\mathsf{P}$ requests. No other interaction with the functionalities is needed. Suppose by contradiction that there exists an environment $E$ that distinguishes this ideal world from the real world, we now construct an adversary that contradicts Definition 6.2. Such an adversary works as follows.

- Let $\mathcal{F}^{\mathsf{P},\mathsf{prf}}_{\mathtt{URF},\mathcal{X},\mathcal{Y}}$ be the functionality used to initialize the ideal/real-world. Upon receiving $I := (\textsc{input}, \mathrm{sid}, y)$ with $y := (\textsc{Initialize}, \mathrm{sid})$ query the oracle KeySetup($\mathsf{prf}$).
- Upon receiving $I := (\textsc{update}, \mathrm{sid}, \mathcal{F}^{\mathsf{P},\mathsf{prf}_j}_{\mathtt{URF},\mathcal{X},\mathcal{Y}})$ from $E$ set $\mathsf{prf} \leftarrow \mathsf{prf}_j$.
- Upon receiving $I := (\textsc{input}, \mathrm{sid}, y)$ where $y = (\textsc{Eval}, \mathrm{sid}, x)$, check if the pair $(\mathsf{prf}, x, y)$ has been recorded. If this is the case then return $y$, else query $\mathrm{Fn}(\mathsf{prf}, x)$, and upon receiving $y$ from the oracle store $(\mathsf{prf}, x, y)$ and return $y$ to the caller.
- Return whatever $E$ returns.

In this case when our adversary is in $\mathsf{FF.PR.Gm}^0$ then the output of the environment corresponds to the output it returns when interacting with the ideal functionality (and corresponds to the output it returns when interacts in the real world when our adversary is in $\mathsf{FF.PR.Gm}^1$). This concludes the theorem. □

**Acknowledgments.** We thank the anonymous reviewers of Crypto 2023 and TCC 2023 for their insightful feedback that led to various improvements of this work.

The second author is supported by the Sunday Group.

## References

[ABBC10] Tolga Acar, Mira Belenkiy, Mihir Bellare, and David Cash. Cryptographic agility and its relation to circular encryption. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 403–422. Springer, Heidelberg, May / June 2010.

[ACJ17] Prabhanjan Ananth, Aloni Cohen, and Abhishek Jain. Cryptography with updates. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 445–472. Springer, Heidelberg, April / May 2017.

[BBF+16] Karthikeyan Bhargavan, Christina Brzuska, Cédric Fournet, Matthew Green, Markulf Kohlweiss, and Santiago Zanella-Béguelin. Downgrade resilience in key-exchange protocols. In *2016 IEEE Symposium on Security and Privacy*, pages 506–525. IEEE Computer Society Press, May 2016.

[BCH+20] Christian Badertscher, Ran Canetti, Julia Hesse, Björn Tackmann, and Vassilis Zikas. Universal composition with global subroutines: Capturing global setup within plain UC. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part III*, volume 12552 of *LNCS*, pages 1–30. Springer, Heidelberg, November 2020.

[BDGJ20] Colin Boyd, Gareth T. Davies, Kristian Gjøsteen, and Yao Jiang. Fast and secure updatable encryption. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 464–493. Springer, Heidelberg, August 2020.

[BGG94] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography: The case of hashing and signing. In Yvo Desmedt, editor, *CRYPTO'94*, volume 839 of *LNCS*, pages 216–233. Springer, Heidelberg, August 1994.

[BH04] Michael Backes and Dennis Hofheinz. How to break and repair a universally composable signature functionality. In Kan Zhang and Yuliang Zheng, editors, *ISC 2004*, volume 3225 of *LNCS*, pages 61–72. Springer, Heidelberg, September 2004.

[BPR14] Mihir Bellare, Kenneth G. Paterson, and Phillip Rogaway. Security of symmetric encryption against mass surveillance. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 1–19. Springer, Heidelberg, August 2014.

[Can03] Ran Canetti. Universally composable signatures, certification and authentication. Cryptology ePrint Archive, Report 2003/239, 2003. `https://eprint.iacr.org/2003/239`.

[Can20] Ran Canetti. Universally composable security. In *Journal of the ACM, Vol. 67, No. 5, 2020*, 2020.

[CJL+16] Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. Report on post-quantum cryptography. NISTIR 8105, National Institute of Standards and Technology, `https://nvlpubs.nist.gov/nistpubs/ir/2016/nist.ir.8105.pdf`, April 2016.

[CKKZ20] Michele Ciampi, Nikos Karayannidis, Aggelos Kiayias, and Dionysis Zindros. Updatable blockchains. In Liqun Chen, Ninghui Li, Kaitai Liang, and Steve A. Schneider, editors, *ESORICS 2020, Part II*, volume 12309 of *LNCS*, pages 590–609. Springer, Heidelberg, September 2020.

[CMNV22] Suvradip Chakraborty, Bernardo Magri, Jesper Buus Nielsen, and Daniele Venturi. Universally composable subversion-resilient cryptography. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part I*, volume 13275 of *LNCS*, pages 272–302. Springer, Heidelberg, May / June 2022.

[CSW22] Ran Canetti, Pratik Sarkar, and Xiao Wang. Triply adaptive uc nizk. In Shweta Agrawal, editor, *Advances in Cryptology – ASIACRYPT 2022*. Springer Nature Switzerland, 2022.

[DGMT17] Grégory Demay, Peter Gazi, Ueli Maurer, and Björn Tackmann. Per-session security: Password-based cryptography revisited. In Simon N. Foley, Dieter Gollmann, and Einar Snekkenes, editors, *ESORICS 2017, Part I*, volume 10492 of *LNCS*, pages 408–426. Springer, Heidelberg, September 2017.

[DKXY03] Yevgeniy Dodis, Jonathan Katz, Shouhuai Xu, and Moti Yung. Strong key-insulated signature schemes. In Yvo Desmedt, editor, *PKC 2003*, volume 2567 of *LNCS*, pages 130–144. Springer, Heidelberg, January 2003.

[FM18] Marc Fischlin and Sogol Mazaheri. Self-guarding cryptographic protocols against algorithm substitution attacks. In Steve Chong and Stephanie Delaune, editors, *CSF 2018 Computer Security Foundations Symposium*, pages 76–90. IEEE Computer Society Press, 2018.

[Fra06] Matthew K. Franklin. A survey of key evolving cryptosystems. *Int. J. Secur. Networks*, 1(1/2):46–53, 2006.

[GOS12] Jens Groth, Rafail Ostrovsky, and Amit Sahai. New techniques for noninteractive zero-knowledge. *J. ACM*, 59(3), jun 2012.

[JPS13] Tibor Jager, Kenneth G. Paterson, and Juraj Somorovsky. One bad apple: Backwards compatibility attacks on state-of-the-art cryptography. In *NDSS 2013*. The Internet Society, February 2013.

[JT20] Joseph Jaeger and Nirvan Tyagi. Handling adaptive compromise for practical encryption schemes. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 3–32. Springer, Heidelberg, August 2020.

[LT18] Anja Lehmann and Björn Tackmann. Updatable encryption with post-compromise security. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 685–716. Springer, Heidelberg, April / May 2018.

[Mau02] Ueli M. Maurer. Indistinguishability of random systems. In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 110–132. Springer, Heidelberg, April / May 2002.

[MR11] Ueli Maurer and Renato Renner. Abstract cryptography. In Bernard Chazelle, editor, *ICS 2011*, pages 1–21. Tsinghua University Press, January 2011.

[OPM22] David Ott, Kenny Paterson, and Dennis Moreau. Where is the research on cryptographic transition and agility? RWC, 2022. Video of the talk `https://www.youtube.com/watch?v=nF2CB8VD-IA` at 1:03:00.

[PR22] Bertram Poettering and Simon Rastikian. Sequential digital signatures for cryptographic software-update authentication. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen, and Weizhi Meng, editors, *ESORICS 2022, Part II*, volume 13555 of *LNCS*, pages 255–274. Springer, 2022.

[Ras17] Fahmida Y. Rashid. Oracle to java devs: Stop signing jar files with md5. `https://www.infoworld.com/article/3159186/oracle-to-java-devs-stop-signing-jar-files-with-md5.html`, 19 January 2017.

[SBK+17] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full SHA-1. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 570–596. Springer, Heidelberg, August 2017.

[SSA+09] Marc Stevens, Alexander Sotirov, Jacob Appelbaum, Arjen K. Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. In Shai Halevi, editor, *CRYPTO 2009*, volume 5677 of *LNCS*, pages 55–69. Springer, Heidelberg, August 2009.

[Sul09] Bryan Sullivan. Security briefs - cryptographic agility. *MSDN Magazine*, 24(8), August 2009.

[WY05] Xiaoyun Wang and Hongbo Yu. How to break MD5 and other hash functions. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *LNCS*, pages 19–35. Springer, Heidelberg, May 2005.

[WYY05] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 17–36. Springer, Heidelberg, August 2005.

# Supplementary Material

# A Standard Cryptographic Tools used in this Work

## A.1 Digital Signatures

A signature scheme is a triple of PPT algorithms $\mathcal{DS} = (\mathsf{KGen}, \mathsf{Sign}, \mathsf{Ver})$ and is required to satisfy completeness, consistency, and unforgeability. Completeness captures the case that honestly generated signatures can be successfully verified, consistency says that one can associate to each triple $(vk, m, \sigma)$ (consisting of verification key, message, and signature string) a unique verification result. Finally, unforgeability is the property that it is computationally infeasible to generate, for a given honestly generated verification key $vk$, a signature $\sigma$ for a message $m$ unless one has seen a signature on $m$ previously.

In UC, a signature scheme is captured as a UC-protocol $\pi^S_{\mathtt{SIG}}$, where $S$ is the identity of the sender, constructed toward realizing an ideal signature functionality $\mathcal{F}^S_{\mathtt{SIG}}$ (cf. Figure 11 below). This is achieved by restricting the execution of $\mathsf{KGen}$ (and outputting the verification key) and the execution of $\mathsf{Sign}$ to party $S$, whereas any party can execute $\mathsf{Ver}$. The above set of properties is equivalent to the statement that $\pi^S_{\mathtt{SIG}}$ UC-realizes $\mathcal{F}^S_{\mathtt{SIG}}$ (see for example [Can03]). Since we are only interested in UC security, we omit specifying the classical properties formally.

A signature is often used in combination with a public-key infrastructure to verify the association of a public key $vk$ with the entity $S$. This is typically modeled by an ideal PKI functionality that allows a party $S$ to register a public key $vk$, and exports the pair $(S, vk)$ to everyone upon request. We omit the simple specification of this as an ideal functionality and refer to [Can03] for details.

### A.1.1 UC Functionality for Signatures

The signature functionality is given below in Figure 11. It is based on [Can03] where the state variable is made explicit here. We further observe (along the lines noted in [BH04]) that we have to complete the specification by explicitly specifying the correct behavior upon signer-key registration depending on the current state of the functionality, to ensure that the expected ideal unforgeability properties hold no matter when the honest signer is initialized. We note that the respective proofs given in [Can03] or [BH04] do actually imply this particular behavior (as it is an obvious consequence of the unforgeability property of signature schemes) and we just write it out explicitly here.

**Signature Functionality with Sub-Corruptions.** We denote by $\widehat{\mathcal{F}}^S_{\mathtt{SIG}}$ the straightforward generalization of this functionality to sub-corruptions with the behavior that is identical to the above, except that it formally accepts on its backdoor tape sub-corruption queries $(id, P)$ in which case entries of the form $(id, P)$ or $P$ are recorded in $\mathcal{C}$ (and the functionality reacts to either of them). See Section 3.1 for more details.

**Functionality $\mathcal{F}_{\mathtt{SIG}}^{S}$**

- Upon receiving a value $(\textsc{key-gen}, \mathrm{sid})$ from the signer $S$, hand $(\textsc{key-gen}, \mathrm{sid})$ to $\mathcal{A}$. Upon receiving $(\textsc{verification-key}, \mathrm{sid}, vk)$ from $\mathcal{A}$, verify that no entry $(\cdot, m, \cdot, vk, 1)$ is recorded and ignore the reply if there is such an entry. Else, record the pair $(S, vk)$ in $\mathsf{state}_{\mathtt{SIG}}$ and output $(\textsc{verification-key}, \mathrm{sid}, vk)$ to $S$.
- Upon receiving $(\textsc{sign}, \mathrm{sid}, m)$ from party $S$, send $(\textsc{sign}, \mathrm{sid}, m)$ to $\mathcal{A}$. Upon receiving $I = (\textsc{signature}, \mathrm{sid}, m, \sigma)$ from $\mathcal{A}$, verify that no entry $(\cdot, m, \sigma, vk, 0) \in \mathsf{state}_{\mathtt{SIG}}$ is stored and ignore the reply if there is such an entry. Else, send $(\textsc{signature}, \mathrm{sid}, m, \sigma)$ to $S$, and store the entry $(S, m, \sigma, vk, 1)$ in $\mathsf{state}_{\mathtt{SIG}}$.
- Upon receiving a value $(\textsc{verify}, \mathrm{sid}, m, \sigma, vk')$ from some party $P$, hand $(\textsc{verify}, \mathrm{sid}, m, \sigma, vk')$ to the adversary. Upon receiving $(\textsc{verified}, \mathrm{sid}, m, \phi)$ from the adversary do:
  1. If $(S, vk) \in \mathsf{state}_{\mathtt{SIG}} \wedge vk' = vk$ and $(\cdot, m, \sigma, vk, 1) \in \mathsf{state}_{\mathtt{SIG}}$, then set $f = 1$. (This condition guarantees completeness: If the verification key $vk'$ is the registered one and $\sigma$ is a legitimately generated signature for $m$, then the verification succeeds.)
  2. Else, if $(S, vk) \in \mathsf{state}_{\mathtt{SIG}} \wedge vk' = vk$ and $S \notin \mathcal{C}$, and $\neg\exists\sigma' : (m, \sigma', vk, 1) \in \mathsf{state}_{\mathtt{SIG}}$, then set $f = 0$ and record the entry $(\cdot, m, \sigma, vk, 0)$ in $\mathsf{state}_{\mathtt{SIG}}$. (This condition guarantees unforgeability: If $vk'$ is the registered one, the signer is not corrupted, and never signed $m$, then the verification fails.)
  3. Else, if there is an entry $(\cdot, m, \sigma, vk', f')$ stored, then let $f = f'$ . (This condition guarantees consistency: All verification requests with identical parameters will result in the same answer.)
  4. Else, let $f = \phi$ and record the entry $(P, m, \sigma, vk', \phi)$ in $\mathsf{state}_{\mathtt{SIG}}$.

  Send $(\textsc{verified}, \mathrm{sid}, m, f)$ to $P$.
- The functionality follows the standard corruption mode; in particular, the adversary is activated on any input from $P \in \mathcal{C}$ and can decide its return value directly, where $\mathcal{C}$ is the corruption set.

Figure 11: The standard Signature functionality.

## A.2 UC NIZK

A non-interactive zero-knowledge proof system for an NP relation $R$ is comprised of three algorithms $\mathcal{PS} = (\mathsf{Gen}, \mathsf{Prove}, \mathsf{Vrfy})$, where $\mathsf{Gen}$ outputs a common-reference string ($\mathsf{crs}$), $\mathsf{Prove}$ takes as input $\mathsf{crs}$ as well as a statement and a witness $(x, w) \in R$ and outputs a proof $\pi$. $\mathsf{Vrfy}$ takes as input $\mathsf{crs}$ as well as a statement $x$ and a proof $\pi$ and returns a decision bit. We require from a NIZK system that it is complete, sound, and zero-knowledge. The above triple of algorithms can be cast in UC [GOS12, CSW22] and, in its simplest form, be captured analogous to signatures: we assume an ideal process that outputs $\mathsf{crs}$ to all participants, and have a prover $\mathtt{P}$ execute the algorithm $\mathsf{Prove}$, and all other parties execute $\mathsf{Vrfy}$. This protocol is demanded to UC-realize $\mathcal{F}_{\mathtt{NIZK}}^{\mathtt{P}}$. Note that there are several settings of interest that differ in the assumptions that admit UC realization [CSW22]. For example, one could distinguish single prover (per CRS) systems from multiple prover systems, static vs. adaptive security, or even consider a corruption mode that includes subverting the CRS. As we will see later, the exact requirements how to obtain $\mathcal{F}_{\mathtt{NIZK}}$ is not of importance to our update system. We will show how to formalize all the above corruption modes including subversion and CRS corruption in later sections. For completeness, the functionalities are given in Figure 12.

### A.2.1 UC Functionality for NIZK

The functionality for UC NIZK is given in Figure 12. We formulate it with respect to a known prover identity for simplicity. However, other variants are derived easily [GOS12, CSW22]. We again consider the formal extension to sub-corruption by $\widehat{\mathcal{F}}^{\mathtt{P}}_{\mathtt{NIIZK}}$.

**Functionality $\mathcal{F}^{\mathtt{P}}_{\mathtt{NIZK}}$**

The functionality is parameterized by an NP relation $R$.

- Upon receiving $(\textsc{prove}, \mathrm{sid}, x, w)$ from $\mathtt{P}$: if $R(x, w) = 1$ then do the following: send $(\textsc{prove}, \mathrm{sid}, x)$ to $\mathcal{A}$; Upon receiving $(\textsc{proof}, \mathrm{sid}, x, \pi)$ from $\mathcal{A}$ store $(\mathtt{P}, x, w, \pi)$ in $\mathsf{state}_{\mathtt{NIZK}}$ and output $(\textsc{proof}, \mathrm{sid}, x, \pi)$ to $\mathtt{P}$.
- Upon receiving $(\textsc{verify}, \mathrm{sid}, x, \pi)$ from a party $P$: if $(\cdot, x, w, \pi) \in \mathsf{state}_{\mathtt{NIZK}}$ return $(\textsc{verified}, x, \pi, R(x, w))$ to $P$. Else send $(\textsc{verify}, \mathrm{sid}, P, x, \pi)$ to $\mathcal{A}$ and parse the reply from $\mathcal{A}$ as $(\textsc{witness}, \mathrm{sid}, x, w)$, store $(P, x, w, \pi)$ in $\mathsf{state}_{\mathtt{NIZK}}$ and return $(\textsc{verified}, x, \pi, R(x, w))$ to $V$.
- The functionality follows the standard corruption mode (the adversary is activated on any input from $P \in \mathcal{C}$ and can decide its return value directly, where $\mathcal{C}$ is the corruption set).

Figure 12: The standard NIZK functionality.

When realizing the sub-corruptible version $\widehat{\mathcal{F}}^{\mathtt{P}}_{\mathtt{NIIZK}}$ of $\mathcal{F}^{\mathtt{P}}_{\mathtt{NIZK}}$, the sub-corruption stems not only from using possibly subverted algorithms to realize it, but from possible subversion or leakage of the CRS (such as corrupting the CRS ceremony). The CRS for proof systems can thus be generalized to include various forms of corruption as we show below:

**Functionality $\mathcal{F}^{\mathsf{Gen}}_{\mathsf{CRS}}$**

On the first activation, the functionality samples $r \xleftarrow{\$} \{0,1\}^\lambda$ and runs $\mathsf{Gen}(1^\lambda; r)$ thus obtaining $\mathsf{crs}$. Then it stores $\mathsf{crs}$ and $r$.

- Upon receiving $(\textsc{get-crs}, \mathrm{sid})$ from any party $P \in \mathcal{P}$ send $\mathsf{crs}$ to $p$.
- Upon receiving $(\textsc{sub-corrupt}, \mathrm{sid})$[a] from $\mathcal{A}$ mark all subroutines $(\mathcal{F}_{\mathsf{CRS}}, \mathrm{sid}, \cdot)$ of the party set as corrupted and reveal the random tape to the adversary.

[a] In case of static corruption, this capability must be exercised before anyone obtains the CRS. In case of an incorruptible CRS, this input is ignored.

Figure 13: A CRS functionality that captures static, adaptive, and incorruptible corruption modes.

## A.3 Pseudo-Random Functions

**Pseudo-random functions.** A pseudo-random function ensemble is a set of functions $\{f_s\}_{s \in \{0,1\}^\kappa}$ (with domain $\mathcal{X}_\kappa = \{0,1\}^{\ell(\kappa)}$ and range $\mathcal{Y}_\kappa = \{0,1\}^{\ell(\kappa)}$ for some function $\ell$ and security parameter $\kappa$) associated with two efficient algorithms $(\mathsf{KGen}, \mathsf{Eval})$ such that $\mathsf{Eval}(s, x) = f_s(x)$ and such that the function ensemble $\{F_\kappa\}_{\kappa \in \mathbb{N}}$ is pseudo-random, where $F_\kappa := f_{\mathsf{KGen}(1^\kappa)}$. Again we are interested in the UC formalization of this guarantee. The standard way to capture pseudo-random

functions in a composable framework[4] is to first define a protocol $\pi^{\mathtt{P}}_{\mathtt{URF}}$ which, running in some session sid (and on some security parameter $\kappa$), upon initialization executes $s \xleftarrow{\$} \mathsf{KGen}(1^\kappa)$ and subsequently accepts inputs $(\textsc{Eval}, \mathrm{sid}, x)$ upon which it returns $\mathsf{Eval}(s, x)$. Second, this protocol must be indistinguishable form an ideal uniformly random function as specified in Figure 14, i.e., no efficient environment has a non-negligible advantage (in the security parameter $\kappa$) in distinguishing an execution of the real protocol $\pi^{\mathtt{P}}_{\mathtt{URF}}$ and an execution of the ideal protocol, i.e., with $\mathscr{F}_{\mathtt{URF},\mathcal{X}_\kappa,\mathcal{Y}_\kappa}$. Note that we will again leave the security parameter implicit for notational simplicity.

### A.3.1 UC Functionality for Random Functions

For completeness, we provide here the UC formalization of a random function in Figure 14 with (concrete) domains $\mathcal{X}$ and $\mathcal{Y}$.

**Functionality $\mathcal{F}^{\mathtt{P}}_{\mathtt{URF},\mathcal{X},\mathcal{Y}}$**

- Upon receiving $(\textsc{Initialize}, \mathrm{sid})$ from $\mathtt{P}$: if not yet initialized, initialize a new table $T$ for which $T[x] = \perp$ for all $x \in \mathcal{X}$. Store $T$ in $\mathsf{state}_{\mathtt{URF}}$.
- Upon receiving $(\textsc{Eval}, \mathrm{sid}, x)$ from $\mathtt{P}$: ignore the request if the party is not initialized yet. Else, if $T[x] \neq \perp$ return $T[x]$ and otherwise choose $y \xleftarrow{\$} \mathcal{Y}$, update $T[x] \leftarrow y$, store the updated table in $\mathsf{state}_{\mathtt{URF}}$ and return the function value $T[x]$. (Ignore the request if already initialized.)
- The functionality follows the static corruption mode (the adversary either takes full control of $\mathtt{P}$ upon the first activation, or $\mathtt{P}$ remains honest).

Figure 14: UC functionality capturing a random function.

## A.4 Hash Functions

A hash function is a pair of probabilistic polynomial- time algorithms $(\mathsf{Gen_H}, H)$ satisfying the following:

- $\mathsf{Gen_H}$ is a probabilistic algorithm which takes as input a security parameter $1^\lambda$ and outputs a key $k$. We assume that $1^\lambda$ is implicit in $k$.
- There exists a polynomial $\ell$ such that $H$ takes as input a key $k$ and a string $x \in \{0,1\}^\star$ and outputs a string $H_k(x) \in \{0,1\}^{\ell(\lambda)}$.

**The collision finding experiment $\mathsf{HashColl}_{\mathcal{A}}(\lambda)$**

1. A key $k$ is generated by running $\mathsf{Gen_H}(1^\lambda)$.
2. The adversary $\mathcal{A}$ is givens $k$ and outputs $x, x'$.
3. The output of the experiment is defined to be 1 if and only if $x \neq x'$ $H_k(x) = H_k(x')$ In such a case we say that $\mathcal{A}$ has found a collision.

[4] In the context of composable PRF security as considered in this work, we focus on the case of static UC corruptions for simplicity.

We say that a hash function $\Pi = (\mathsf{Gen_H}, H)$ is collision resistant if for all probabilistic polynomial time-adversaries $\mathcal{A}$ there exists a negligible function $\nu$ such that $\Pr[\mathsf{HashColl}_{\mathcal{A}}(\lambda) = 1] \leq \nu(\lambda)$. We note in passing that in a concrete treatment, one often constructs from an assumed adversary against a cryptosystem an explicit collision-finding algorithm against an underlying hash-function, which is more in line with how hash functions are used in practice.

# B On the Concept of Input-Output Behavior of Functionalities

Sometimes it is useful to talk about the abstract concept of input-output behavior of machines, in particular for the main ITIs of a UC execution of a protocol $\mu$, instead of the very low-level execution model. Here we briefly sketch the basics and refer to the behavior-based models for details and comparisons [Mau02, MR11, DGMT17]. Let $\mathit{config}_i$ denote the configuration of a system of ITIs after the $i$th step of an execution. Transitions between configurations can be modeled generally by conditional probability distributions (defined on the configuration space $C$) $p^{\mu,\mathcal{Z},A}_{C_i|C_{i-1}}(\mathit{config}_i|\mathit{config}_{i-1})$, defined for all configurations $\mathit{config}_{i-1}$ reachable from the initial configuration $\mathit{config}_0$. Clearly, $p^{\mu,\mathcal{Z},A}_{C_i|C_{i-1}}(\mathit{config}_i|\mathit{config}_{i-1})$ can only be positive for valid transitions between two configurations defined by the machine model.[5] Based on this, the transition probabilities $p^{\mu,\mathcal{Z},A}_{C'|C}(\mathit{config}'|\mathit{config})$ between any two configurations are defined (note that we just deal with finite executions) for any configuration $\mathit{config}$ reachable from $\mathit{config}_0$ in finite steps. This further induces the abstract input-output behavior of the *main ITIs* $M = ((\mathit{sid}, P), \mu)$ of a protocol session, i.e., the distribution of return values upon receiving input $x$ at a certain point in the execution. Let $\mathit{config}$ be the configuration in which $M$ is activated and the content of its input tape is $x$ (written by the environment). The distribution of return values is given by the function $p^{M,\mathcal{Z},A}_{Y|(X,C)}(y|(x, \mathit{config})) = \sum_{\mathit{config}'} p^{\mu,\mathcal{Z},A}_{C'|C}(\mathit{config}'|\mathit{config})$, where the sum ranges over all configurations ending in a configuration $\mathit{config}'$ that activates $\mathcal{Z}$ on input $y_x$ (on the subroutine output tape of $\mathcal{Z}$) as the answer to $x$. Note that the concept "the answer $y_x$ to $x$" is only well-defined if there is a mapping between input and output values as part of the execution model. While it is out of the scope of this paper to define such a mapping generically, we note that such a concept is rather intuitive and achievable, e.g. by equipping any input with a unique (per machine) ID and defining that an output is only an answer to some input if the output quotes that ID explicitly. Note that such a mapping is often either implicitly assumed in the UC literature or it is assumed that a calling protocol is able to make the assignment of inputs to outputs.

Finally, observe that when two UC executions are indistinguishable, i.e., $\textsc{exec}_{\pi,\mathcal{A},\mathcal{Z}} \approx \textsc{exec}_{\phi,\mathcal{S},\mathcal{Z}}$, then this in particular implies that the input-output behavior of a machine $M = ((\mathrm{sid}, P), \pi)$ must be *consistent* with the input-output behavior of the corresponding machine $M' = ((\mathrm{sid}, P), \phi)$ in the following sense: given a strategy $S$ of machine $\mathcal{Z}$ (interacting with the main ITIs of the system) and any configurations $\mathit{config}^{(\pi)}$ and $\mathit{config}^{(\phi)}$ reachable (and induced by $S$) with non-negligible probability, it holds that the two associated distributions $p^{M,\mathcal{Z},A}_{Y|(X,C)}(y|(x, \mathit{config}^{(\pi)}))$ and $p^{M',\mathcal{Z},A}_{Y|(X,C)}(y|(x, \mathit{config}^{(\phi)}))$ must be computationally close (where $x$ is the provided input to the machine).

[5] Recall from [Can20] that the execution of a system of ITIs starts with $\mathcal{Z}$ that invokes other machines, that either execute the instructions of the protocol $\mu$ or the adversarial (or simulation) strategy $A$. This is sufficient to derive all possible valid transitions between configurations. A configuration corresponds to all machines in their respective configuration. The initial configuration $\mathit{config}_0$ in UC is well-defined for any execution.

# C Generic Facts about Update Systems

In this section, we employ the abstract model to express some simple facts about UC updates. We consider the class of functionalities that have the property that their behavior of $\mathcal{F}_i$ is independent of the caller's code, i.e., their behavior is identical when called by external ITI $eid = (\rho, sid, pid)$ or $eid' = (\rho', sid, pid)$.

**Lemma C.1.** *Let $\mathscr{F} = \{\mathcal{F}_0, \mathcal{F}_1, \ldots, \mathcal{F}_n\}$ be a set of functionalities with the property that the behavior of $\mathcal{F}_i$ is independent of the caller's code. If we require that* $\texttt{UpPred} \equiv 1$ *and* $\texttt{SttUp} \equiv \emptyset$ *then there is a simple update mechanism for any such class of functionalities.*

*Proof.* We define a simple protocol $\pi$ with access to hybrid functionalities $\mathcal{F}_0, \mathcal{F}_1, \ldots, \mathcal{F}_n$ and performs the update between them. The protocol works as follows for each party $P$ (that encodes some session id *sid* and party id *pid*). On input $I = (\textsc{input}, \mathrm{sid}, x)$ from a party with identity $(\cdot, sid, pid)$, party $P$ relays this input $x$ to the initial instance of $\mathcal{F}_0$ (sub-session can simply be sid). Upon any subroutine output by $\mathcal{F}_0$ send the output to the respective machine. (Note that $P$ does not need to store any information about the input $x$ or any output.)

On input $I := (\textsc{update}, \mathrm{sid}, \mathcal{F})$ for $\mathcal{F} \in \mathscr{F}$, simply output $(\mathsf{Success}, \mathrm{sid}, \mathcal{F})$. Upon subsequent inputs, the party behaves as with the previous case, except that inputs are relayed only to the new instance $\mathcal{F}$ (in session sid) and only outputs from this instance are provided to the callers.

Sketching a simulator $\mathcal{S}$ for this protocol is straightforward: whenever activated by functionality $\mathsf{U}^{\mathscr{F}}_{\texttt{SttUp},\texttt{UpPred}}$ with an output of a simulated instance of a functionality $\mathcal{F}_i$, then $\mathcal{S}$ relays this output to $\mathcal{Z}$ as coming from $\mathcal{F}_i$. On input $(\textsc{update_notification}, \mathsf{UpdateReq}_{\mathcal{F}}, P, \mathcal{F})$, where $\mathcal{F} \in \mathscr{F}$, then $\mathcal{S}$ sends $(\textsc{GrantUpdate}, \mathrm{sid}, P, \mathrm{sid}, \mathcal{F})$ to $\mathsf{U}^{\mathscr{F}}_{\texttt{SttUp},\texttt{UpPred}}$. Complementary, whenever activated with input destined for a backdoor tape, $\mathcal{S}$ relays this to the corresponding functionality.

It is easy to see that this behavior is indistinguishable from the real world. Note that since the behavior of functionalities in $\mathsf{U}^{\mathscr{F}}_{\texttt{SttUp},\texttt{UpPred}}$ is assumed to be independent of the code of the caller, the overlay introduced in the real world will not be noticeable by any environment. Finally, since $\texttt{UpPred} \equiv 1$ any update instructed by the simulator will succeed just as in the real world. □

We next study what the hard problem behind updates is in general. In view of Section B, let us state a simple definition.

**Definition C.2.** An update protocol is trivial if with any UC environment and adversary, no party ever concludes an update with non-negligible probability.

The following lemma captures our intuition that the hard part of designing an update protocol is trying to achieve that the behavior after the update must be consistent with a certain preserved state that is a function of the execution prefix. This is as natural general observation and applies to any functionality. In more detail, it says that if a set of parties $\mathcal{P}$ perform an update (to implement a new functionality $\mathcal{F}_1$), then the observable behavior of these parties must be consistent with the behavior of $\mathcal{F}_1$ with a well-defined initialized state at the time, for example, where the first honest party completes the update. Two executions are consistent if they have indistinguishable behavior (despite different code; cf. Section B for more details). For example, when updating a signature scheme, the difficult part is to agree on the set of previously signed messages w.r.t. which consistency holds. Otherwise, the behavior is not consistent for at least one party $P \in \mathcal{P}$ as they have potentially incompatible sets of verified messages, and can thus never be considered to be in the same "updated instance" (an environment would have no problem in detecting the mismatch).

**Lemma C.3.** *Let $\pi$ be a non-trivial update protocol for $\mathscr{F} = \{\mathcal{F}_0, \mathcal{F}_1\}$. The input-output behavior, for any party $P$ that completes an update from an instance of $\mathcal{F}_0$ to an instance of $\mathcal{F}_1$, is consistent*

*with an ideal execution (w.r.t. some simulator) of $\mathcal{F}_1$, where $\mathcal{F}_1$ must be initialized with a well-defined value* $\mathsf{state}$*, which is a function of the execution prefix until the point when that party $P$ completes the update.*

*Proof Sketch.* Since $\pi$ is an update protocol for $\mathscr{F}$, we have that before the first update $\pi \equiv \mathcal{F}_0$. Let $\mathcal{Z}$ denote the environment. Since the update is assumed to be non-trivial and UC-realizes $\mathsf{U}^{\mathscr{F}}_{\mathtt{SttUp},\mathtt{UpPred}}$, there is a sequence of instructions made by $\mathcal{Z}$ in the real world such that the execution prefix records and update success for $P$. Let further $\mathcal{S}$ be the assumed simulator in the ideal execution. Clearly, since both worlds are assumed to be indistinguishable, the ideal world must output an update success for $P$ based on the same strategy implemented by $\mathcal{Z}$ (cf. Section 3.1, the strategy corresponds to a sequence of conditional probability distributions that fully describes the environments behavior). At this point of the first update in the ideal world, $\mathtt{UpPred}(.) = 1$, where the state $\mathsf{state}_0$ of $\mathcal{F}_0$ does not change because the input $I := (\textsc{GrantUpdate}, \mathrm{sid}, P, \mathrm{ssid}', \mathcal{F}')$ does not have any side-effect. Thus if $P$ is the first party to complete the update, then both, $\mathsf{state}_0$ and $\mathsf{state}_1 := \mathtt{SttUp}(\mathsf{state}_0, \cdot)$ are well-defined. In particular, $\mathsf{state}_1$, the initial configuration of $\mathcal{F}_1$ is uniquely defined in the ideal world. From Section 3.1 we know that if the real and ideal-world executions are indistinguishable, this implies that the behavior of the protocol machine of party $P$ must be consistent with the behavior of party $P$ in the ideal world. By definition of $\mathsf{U}^{\mathscr{F}}_{\mathtt{SttUp},\mathtt{UpPred}}$, the inputs of $P$ (after the update) are provided to $\mathcal{F}_1$ and $P$ only receives outputs from $\mathcal{F}_1$ (after the update, possibly influenced by the assumed simulator $\mathcal{S}$), where $\mathcal{F}_1$ was initialized with a compatible $\mathsf{state}_1$. The same reasoning applies to any further honest party that completes the update. Again, the execution prefix defines at the point the update completes defines the state of $\mathcal{F}_1$ relative to which all parties interacting with $\mathcal{F}_1$ must have a consistent behavior. □

# D Message Registry

A Message Registry is a primitive that enables a party to deposit messages into a "container" structure $A$; depositing any message produces an associated membership witness $w$, so that the primitive's verification algorithm outputs 1 if and only if the element is in the container. Formally, the primitive is composed of the following three algorithms $(\mathsf{Gen}, \mathsf{Add}, \mathsf{Vmr})$.

- $\mathit{trap}, A \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$. We call *trap* the trapdoor and $A$ the container.
- $A', w \xleftarrow{\$} \mathsf{Add}(\mathit{trap}, A, x)$ with $x \in \{0,1\}^\lambda$
- $0/1 \xleftarrow{\$} \mathsf{Vmr}(A, w, x)$

**Correctness.** Informally, correctness requires that the verification outputs 1 as long as a given element with its witness is in a container. Let $(\mathit{trap}, A) \xleftarrow{\$} \mathsf{Gen}(1^\lambda), A_1 := A$. For any $n \in \mathbb{N}$, $X := (x_1, \ldots, x_n) \in \{\{0,1\}^\lambda\}^n$. $\forall i \in [n](A_i, w_i) \xleftarrow{\$} \mathsf{Add}(\mathit{trap}, A_{i-1}, x_i)$ and for any $i \leq j \in [n]$

$$\Pr\left[1 \xleftarrow{\$} \mathsf{Vmr}(A_j, w_i, x_i)\right] \geq 1 - \mathrm{negl}(\lambda)$$

**Security.** We start by defining the oracle $\mathcal{O}(\cdot)$. $\mathcal{O}$ manages the parameter $\mathsf{lastA}$ initialized to $A$, where $A$ is the second output of the generation algorithm $\mathsf{Gen}$. On input a value $x \in \{0,1\}^\lambda$, $\mathcal{O}$ adds $x$ to set of queries $\mathsf{Q}$, computes $A', w \xleftarrow{\$} \mathsf{Add}(\mathit{trap}, A, x)$ sets $\mathsf{lastA} \leftarrow A'$ and returns $\mathsf{lastA}$.

We say that a scheme is secure if the following holds.

$$\Pr\left[\begin{array}{c}(trap, A) \xleftarrow{\$} \mathsf{Gen}(1^\lambda), (x, w) \leftarrow \mathcal{A}^{\mathcal{O}(\cdot)}(A) : \\ 1 \xleftarrow{\$} \mathsf{Vmr}(\mathsf{lastA}, w, x) \text{ and } x \notin \mathsf{Q}\end{array}\right] \leq \mathrm{negl}(\lambda)$$

**Instantiating a Message Registry.** If we are not interested in how the length of the witnesses and of the container grows, then we can information theoretically instantiate a *message registry* scheme. The construction is straightforward. The generation algorithm outputs no trapdoor (we denote that by $trap := \perp$) and an empty set $A = \varnothing$. $\mathsf{Add}$, on input $trap$, $A$ and an element $x \in \{0,1\}^\lambda$, add $x$ to the set $A$ and returns the update set. The witness $w$ is $\perp$. The algorithm $\mathsf{Update}$, on any input return $\perp$. The algorithm $\mathsf{Vmr}$, on input a set $A$, a witness $w$ and $x$, ignores $w$ and return $x \in A$.

The correctness of the scheme comes by inspection. The security also follows immediately since a set is fully defined by the elements that it contains. Hence, given a set $\mathsf{lastA}$, the adversary has no way to pretend that an *element* $x$ belongs to $A$ when $x$ is not already part of $\mathsf{lastA}$. On the other hand, if $x \notin \mathsf{lastA}$, and the only way to prevent that $x \in \mathsf{lastA}$ is by adding $x$ to $\mathsf{lastA}$. But his would just yield to a set $\mathsf{lastA}' \neq \mathsf{lastA}$.

In practice, we might be interested in realizing a message registry scheme that enjoys a form of *succinctness*. In such a scheme the length of the witness or the length of a container does not grow linearly with the number of elements added to the container via $\mathsf{Add}$. It is easy to verify that by relying on a digital signature scheme, we can derive such scheme with computational security. In this instantiation, the container remains fixed to the public-key of the digital signature. The $\mathsf{Add}$ algorithm signs the message equating the witness to the signature. Verification simply verifies the digital signature. It is easy to verify that correctness is implied by the correctness of the digital signautre, while security is implied by unforgeability.