Universally Composable Accumulators

Foteini Badimtsi[1][0000−0003−3296−5336], Ran Canetti[2][0000−0002−5479−7540], and Sophia Yakoubov[2][0000−0001−7958−8537]

[1] George Mason University, Fairfax, VA, USA foteini@gmu.edu
[2] Boston University, Boston, MA, USA
{canetti,sonka}@bu.edu

Abstract. Accumulators, first introduced by Benaloh and de Mare (Eurocrypt 1993), are compact representations of arbitrarily large sets and can be used to prove claims of membership or non-membership about the underlying set. They are almost exclusively used as building blocks in real-world complex systems, including anonymous credentials, group signatures and, more recently, anonymous cryptocurrencies. Having rigorous security analysis for such systems is crucial for their adoption and safe use in the real world, but it can turn out to be extremely challenging given their complexity.

In this work, we provide the first universally composable (UC) treatment of cryptographic accumulators. There are many different types of accumulators: some support additions, some support deletions and some support both; and, orthogonally, some support proofs of membership, some support proofs of non-membership, and some support both. Additionally, some accumulators support public verifiability of set operations, and some do not. Our UC definition covers all of these types of accumulators concisely in a single functionality, and captures the two basic security properties of accumulators: correctness and soundness. We then prove the equivalence of our UC definition to standard accumulator definitions. This implies that existing popular accumulator schemes, such as the RSA accumulator, already meet our UC definition, and that the security proofs of existing systems that leverage such accumulators can be significantly simplified.

Finally, we use our UC definition to get simple proofs of security. We build an accumulator in a modular way out of two weaker accumulators (in the style of Baldimtsi *et al.* (Euro S&P 2017), and we give a simple proof of its UC security. We also show how to simplify the proofs of security of complex systems such as anonymous credentials. Specifically, we show how to extend an anonymous credential system to support revocation by utilizing our results on UC accumulators.

1 Introduction

Accumulators, first introduced by Benaloh and de Mare [Bd94], are compact representations of arbitrarily large sets. Despite being small — ideally constant-size relative to the size of the set they represent! — they enable verification of statements about the set. Given a *membership witness* for some object x together with the accumulator, anyone can verify that x is in the accumulated set. If the accumulator is a *universal* accumulator [LLX07], it also supports *non-membership* witnesses that can be used to verify that elements are not in the accumulated set. Typically, an accumulator is owned by an entity called an *accumulator manager* who can add elements to (and, if the accumulator is *dynamic* [CL02], remove elements from) the set. If the accumulator is *strong* [CHKO08], even a corrupt accumulator manager cannot forge a proof of (non-)membership.

Many crucial primitives are actually special cases of accumulators. For instance, digital signatures are accumulator schemes, where the signature verification key is the accumulator representing the set of signed messages, and the signatures are membership witnesses. The owner of the signing key is the accumulator manager, and she can add elements to the set by signing them. Of course, she cannot *un*-sign elements (without publishing a revocation list, which is not constant in size), and she cannot produce a proof that a given element has not been signed, so this accumulator is neither dynamic nor universal. She can also always prove the membership of arbitrary elements, so this accumulator is not strong.

Another example of an accumulator is a Merkle hash tree. The tree root is the accumulator representing the set of leaf nodes, and the authenticating paths through the tree are membership witnesses. This accumulator supports both element addition and deletion, but when either of those events occur, all existing witnesses must be updated, requiring total work that is linear in the number of member elements. In many situations, this is prohibitively inefficient. The Merkle hash tree accumulator is strong, because all additions and deletions are publicly verifiable (by means of re-execution). Though the intuitive Merkle hash tree accumulator does not support proofs of non-membership, it can be modified to be universal [CHKO08].

One construction of a universal, dynamic (but not strong) accumulator with efficient update algorithms is the RSA accumulator. It is the original accumulator introduced by Benaloh and de Mare [Bd94], augmented with dynamism by Camenisch and Lysyanskaya [CL02], and with universality by Li, Li and Xue [LLX07]. It is one of the most popular accumulator constructions because of its compactness and efficiency.

Although accumulators are frequently analyzed as stand-alone primitives, they are *almost exclusively used as building blocks* in real-world complex systems, including anonymous credentials [CL02,Ngu05,CKS09,BCD$^+$17], group signatures [CL02] and, more recently, anonymous cryptocurrencies [MGGR13]. Having rigorous security analysis for such systems is crucial for their adoption and safe use in the real world, but it can turn out to be extremely challenging given their complexity. When a system consists of multiple building blocks, even if each one of them is proven secure independently, the security analysis of the whole needs to be done from scratch.

Universal Composability. Universally Composable (UC) security [Can01], addresses this problem. Any protocol that has been shown to be UC-secure will maintain its security properties even when it is used concurrently with other arbitrary protocols as part of a larger system. This allows one to formally argue about the security of a complex scheme in a much simpler and cleaner way, as long as all the protocols used within it have already been proven to be UC-secure.

Showing that a protocol is UC-secure consists of two steps. First, we write out a set of instructions called the *ideal functionality*, which define how we would instantiate the primitive if we had an incorruptible third party to delegate its operation to. Second, we show that any attack an adversary carries out against the protocol, it can also carry out against the ideal functionality. This is done by arguing that for any efficient adversary and environment (which sets all parties' inputs, receives all parties' outputs and additionally receives information from the adversary), there exists a simulator such that the environment cannot tell the difference between interacting with the protocol and adversary, and interacting with the ideal functionality and simulator. This proves that any time it suffices to use our ideal functionality within a larger system, we can replace it with our protocol and the system will remain secure.

The modularity and the strong security guarantees provided by UC suggest that protocols should always be designed and proven secure in the UC framework. However, this is only the case for a small fraction of proposed cryptographic schemes. One roadblock to using the UC framework is that not all commonly used sub-protocols have UC definitions and proofs. Some such sub-protocols have already been defined and analyzed in the UC framework (e.g. digital signatures [Can01,Can04], zero-knowledge proofs [CKS11], etc.), but others have not. Cryptographic accumulators are one example of a very common primitive that has never been considered in the context of UC security.

Our Results. In this work, we make the following contributions:

1. We provide the first UC definition (ideal functionality) for cryptographic accumulators. There are many functionality flavors of accumulators: accumulators might support only additions, only deletions or both, and they might support proofs of membership, proofs of non-membership, or both. Our UC definition covers all of these possibilities in a modular way.
2. We then prove the equivalence of our UC definition to standard accumulator security definitions. This implies that existing secure accumulator constructions — such as the RSA accumulator [Bd94,CL02,LLX07] — are UC secure.
3. Finally, we discuss how our UC definition simplifies the proofs of security for schemes that use accumulators as a building block. First, we build an accumulator out of two weaker accumulators (as in [BCD$^+$17], but with stronger privacy properties), and give a simple UC proof of security for that composite accumulator, which we call Braavos$'$. Then, we consider how UC simplifies proofs of security in more complex systems such as anonymous credentials.

Note that when defining a new ideal functionality, there are two possible scenarios: either existing constructions can be proven to securely realize the new functionality (as with digital signatures [Can04]), or new constructions must be developed (as with commitment schemes [CF01]). Our second contribution shows that our accumulator functionality is in the first scenario; popular, existing accumulator constructions already satisfy it. This greatly simplifies the security analysis of existing and future systems that use cryptographic accumulators as a building block.

Informally, two classical properties are considered for cryptographic accumulators. The first is correctness: for every element inside (or outside, for negative accumulators) the accumulated set, an honest witness holder can always prove membership (or non-membership, for negative accumulators) in the set. The second is soundness: for every element outside (or inside, for negative accumulators) the accumulated set, it is infeasible to prove membership (or non-membership, for negative accumulators).

Our ideal functionality is different than most ideal functionalities in that it requires as input from the simulator all of the accumulator algorithms (as previously done in the context of digital signatures [Can04]). This is actually a very intuitive way to build an ideal functionality, since it only deviates from the algorithm outputs when necessary for correctness or soundness. We explain this in more detail in Sec. 3.

We chose not to incorporate secrecy or privacy requirements into our ideal functionality since they depend on specific applications and vary considerably; thus, they are best made separately, as an additional "layer" on top of the basic correctness guarantees captured in this work. Additionally, privacy-aware constructions often use accumulators and privacy-enhancing mechanisms (such as zero-knowledge proofs) as two separate modules, making the formalization here more conducive to modular analysis. We exemplify this point by sketching a modular analysis of the Baldimtsi *et al.* [BCD$^+$17] construction of revocable anonymous credentials from zero knowledge proofs and accumulators.

Outline. We start by setting notation and presenting classical accumulator definitions (but with a twist) in Section 2. Then, in Section 3, we give an ideal UC functionality for accumulators that encompasses both of the properties listed above. In Section 4, we argue that any accumulator that has these properties meets our UC definition, and vice versa. Finally, in Section 5 we discuss how our UC definition of accumulators would simplify the security proof of an existing complex system like anonymous credentials.

1.1 Accumulator Applications

To showcase the importance of a UC analysis for cryptographic accumulators we briefly discuss a few of the most interesting systems that use accumulators as a main building block. The security analysis of all the following systems would be much simpler when the underlying accumulator is UC-secure.

Access Control. Authentication of users is vital to most of the electronic systems we use today. It is usually achieved by giving the user a token, or *credential*, that the user must present to prove that she has permission to access a service. A naive construction for an access control system is to maintain a whitelist of authorized users (i.e., by storing their credentials). Whenever a user wants to access the system she just needs to present her credential, and as long as it is on the whitelist, the user will be given access. When a user needs to be revoked, her credential is just removed from the whitelist. Despite its simplicity, such a solution is not practical, since the size of the whitelist will have to grow linearly with the number of participating users.

Cryptographic accumulators enable more efficient access control systems. Instead of keeping a whitelist, an accumulator can be used to maintain the set of authorized users. Whenever a user is given access to the resource, she is given a credential that can be seen as an accumulator membership witness. One possible construction uses the digital signature accumulator together with a blacklist of revoked users, which grows linearly with the number of revocations. This construction is the one most commonly used in public key infrastructures (PKIs), where a certificate revocation list (CRL) that contains the revoked certificates is published periodically. This solution is more efficient, since usually the number of revoked users is much smaller than the number of total users in the system. However, it is still not ideal, since the blacklist can grow to significant size. A dynamic accumulator — which supports both element additions and deletions while remaining small — is a much better solution.

Anonymous Credentials. The inefficiency of the naive whitelist and blacklist solutions for access control becomes even more problematic when anonymity is considered as a goal of the system: if a user wishes to anonymously show that her credential is on a whitelist (or not on a blacklist), then she would have to perform a zero-knowledge proof of membership (or non-membership) which would require cost linear to the size of the corresponding list. Given how expensive zero knowledge proofs usually are, it is important to avoid doing work linear in the number of valid or revoked members in a system. To avoid this inefficiency, *anonymous credentials schemes* (the most prominent solution for anonymous user authentication) make use of dynamic cryptographic accumulators as an essential building block to allow for efficient proofs of membership (and practical user revocation) [CL02,Ngu05,CKS09]. Idemix [CV02], the leading anonymous credential system by IBM, is such an example of an anonymous credential scheme that employs cryptographic accumulators for user membership management [BCD$^+$17].

Cryptocurrencies. As discussed above, when a proof of membership (or non-membership) needs to be done in zero-knowledge, the naive whitelist and blacklist solutions are not realistic. Anonymous cryptocurrencies, like anonymous credentials, require such zero-knowledge proofs. In order to prove that a payment is valid (and is not a double-spend), when a user wishes to spend a coin that she owns, she must first prove that her coin does not belong in a list of previously spent coins. To ensure anonymity, such a proof must be done in zero-knowledge. Universal cryptographic accumulators are used in Zerocoin [MGGR13] to maintain the set of spent coins while enabling efficient zero-knowledge proofs of non-membership.

Group Signatures. Accumulators have been suggested for building other cryptographic primitives such as group signatures. In a group signature scheme, the group manager maintains a list of valid group members, and periodically grants (or revokes) membership. There has been much research on the topic of group signatures, and a number of efficient schemes have been proposed. One of the first practical solutions supporting revocation uses cryptographic accumulators for user revocation (Camenisch and Lysyanskaya [CL02], building on the ACJT group signature scheme [ACJT00]).

2 Revisiting Classical Accumulator Definitions

We first discuss accumulator terminology and notation and review accumulator algorithms. Then, in Section 2.2, we revise the classical accumulator definitions of security to be more modular, and to support a wider range of accumulator functionalities. These changes make the transition to the UC model more clear and natural.

2.1 Notation and Algorithms

An accumulator is a compact representation of a set $S = \{x_1, \dots x_n\}$, which can be used to prove statements about the underlying set. Different accumulator types and properties have been considered in the literature. Here, we use the terminology and definitions of Baldimtsi *et al.* [BCD$^+$17], who provide a modular view of accumulator functionalities. Like them, we consider four basic types of accumulators:

- *Static accumulator*: represents a fixed set.
- *Additive accumulator*: supports only addition of elements to the set.
- *Subtractive accumulator*: supports only deletion of elements from the set.
- *Dynamic accumulator* [CL02]: supports both additions and deletions.

Note that a trivial way to achieve deletions and additions is by re-instantiating the accumulator with the updated set. Although simple, this takes a polynomial amount of time in the number of element additions or deletions which have been performed up until that point. For practical applications a dynamic accumulator should support both additions and deletions in time which is either independent of the number of operations performed altogether, or at least sublinear in this number.

In addition to considering the types of modifications we can make to accumulated sets, we also consider the types of proofs (membership proofs, non-membership proofs, or both) accumulators support.

- *Positive accumulator*: supports membership proofs.
- *Negative accumulator*: supports non-membership proofs.
- *Universal accumulator* [LLX07]: supports both types of proofs.

We consider three types of parties in the accumulator setting. The *accumulator manager* is a special party who is the "owner" of the accumulated set: she creates the accumulator, adds and deletes elements, and creates membership and non-membership witnesses. A *witness holder*, or *user*, is responsible for an accumulated element (i.e. she owns a credential in a system for which an accumulator is used). She is interested in being able to prove the (non-)membership of that element to others, so she maintains the witness for that element, by updating it when/if necessary. Finally, a *verifier* is any third party who is only interested in checking the proofs of (non-)membership (e.g. a gatekeeper checking credentials).

We now describe the algorithms performed by each party, and summarize them in Figure 1. In Figure 2 we summarize the notation used to describe the different accumulator algorithm input and output parameters.

Accumulator Manager Algorithms. The following are algorithms performed by the accumulator manager who creates the accumulator and maintains it as required. If the accumulator is additive, she can add elements to it by calling the Update algorithm with $\mathsf{Op} = \mathsf{Add}$. If the accumulator is subtractive, she can delete elements by calling Update with $\mathsf{Op} = \mathsf{Del}$. If it is dynamic, she can do both. If the accumulator is positive, the accumulator manager can create membership witnesses by calling $\mathsf{WitCreate}$ with $\mathsf{stts} = \mathsf{in}$ (where stts is a variable representing the *status* of an element, which can be in or out of the set); if it is negative she can create non-membership witnesses by calling $\mathsf{WitCreate}$ with $\mathsf{stts} = \mathsf{out}$. If it is universal, she can do both.

- $\mathsf{Gen}(1^\lambda, S_0) \rightarrow (sk, a_0, m_0)$ outputs the accumulator manager's secret key sk, the accumulator a_0 (representing the initial set $S_0 \subseteq D$ of elements in the accumulator, where D is the domain of the accumulator[3]), and an auxiliary value m_0 necessary for the maintenance of the accumulator (i.e. one could think of m_t being the accumulator manager's memory or storage at step t).
- $\mathsf{Update}(\mathsf{Op}, sk, a_t, m_t, x) \rightarrow (a_{t+1}, m_{t+1}, w^x_{t+1}, \mathsf{upmsg}_{t+1})$ updates the accumulator by either adding or deleting an element. If $\mathsf{Op} = \mathsf{Add}$ it adds the element $x \in D$ to the accumulator and outputs the updated accumulator value a_{t+1} and auxiliary value m_{t+1}, as well as the membership witness w^x_{t+1} for x and an update message upmsg_{t+1}, which enables witness holders to bring their witnesses up to date. If $\mathsf{Op} = \mathsf{Del}$ then

[3] The allowable S_0 sets vary from accumulator to accumulator. There are accumulators that support only $S_0 = \emptyset$; others support any polynomial-size S_0, and yet others support any S_0 that can be expressed as a polynomial number of ranges.

it deletes the element x from the accumulator and outputs a_{t+1}, m_{t+1} and upmsg_{t+1} as before, as well as a *non*-membership witness w^x_{t+1}.

– $\mathsf{WitCreate}(\mathsf{stts}, sk, a_t, m_t, x, (\mathsf{upmsg}_1, \ldots, \mathsf{upmsg}_t)) \rightarrow w^x_t$ creates a (non-) membership witness. If $\mathsf{stts} = \mathsf{in}$ it generates a membership witness w^x_t for x, and if $\mathsf{stts} = \mathsf{out}$ it generates a non-membership witness. (Of course, this algorithm should only succeed in generating a valid membership witness if x is actually in the set, and in generating a non-membership witness if x is not in the set.)

Remark 1. The parameters sk, m and upmsg are optional for some accumulator constructions. For instance, in a Merkle hash tree accumulator there is no secret key sk, and in a digital signature accumulator there is no auxiliary value m or update messages upmsg. Notice that the $\mathsf{WitCreate}$ algorithm takes in both the auxiliary value m and the update messages, which seems redundant; after all, the update messages can always be kept as part of m. The reason we provide the algorithm with both arguments is to account for scenarios which do not use any auxiliary storage.

Remark 2. The notion of a public key is absent on the above definition. One can consider the accumulator value a to be the “public key” of the scheme, since it is used for verification. In fact, in the digital signature accumulator construction, the public verification key is equal to the accumulator value. However, unlike a typical public key, the accumulator value can evolve over time.

Witness Holder Algorithms. Witness holders are interested in proving the (non-)membership of certain elements, and thus maintain witnesses for those elements. They use a witness update algorithm WitUp to sync their witnesses with the accumulator when additions or deletions occur.

– $\mathsf{WitUp}(\mathsf{stts}, x, w^x_t, \mathsf{upmsg}_{t+1}) \rightarrow w^x_{t+1}$ updates the membership witness for element x (if $\mathsf{stts} = \mathsf{in}$) or the non-membership witness if $\mathsf{stts} = \mathsf{out}$. The updates use the update messages upmsg, which contain information about changes to the accumulator value (e.g. that a given element was addedv, what the new accumulator value is, etc).

Verifier/Third Party Algorithms. The last category of accumulator users are the *verifiers* (or third parties) who are only interested in checking proofs of (non-)membership. They do so by calling the $\mathsf{VerStatus}$ algorithm.

– $\mathsf{VerStatus}(\mathsf{stts}, a_t, x, w^x_t) \rightarrow \phi$ checks whether the membership witness (if $\mathsf{stts} = \mathsf{in}$) or the non-membership witness (if $\mathsf{stts} = \mathsf{out}$) for element x is valid; it returns $\phi = 1$ if it is, and $\phi = 0$ if it is not.

If the accumulator is *strong* (Definition 4), the accumulator should be secure even against a cheating accumulator manager. That is, all modifications that an accumulator manager makes to the accumulator should be publicly verifiable. The differences in the algorithms are as follows: (a) Gen and Update also output a value v, which essentially is a proof that an accumulator was created/updated correctly. (b) Additional verification algorithms VerGen and $\mathsf{VerUpdate}$ can be used to check these proofs.

Algorithm	Inputs	Outputs
Accumulator Manager Algorithms		
Gen	$1^\lambda, S_0$	sk, a_0, m_0, v
Update	$\mathsf{Op}_t, sk, a_t, m_t, x$	$a_{t+1}, m_{t+1}, w^x_{t+1}, \mathsf{upmsg}_{t+1}, \mathsf{v}_{t+1}$
WitCreate	$\mathsf{stts}, sk, a_t, m_t, (\mathsf{upmsg}_1, \ldots, \mathsf{upmsg}_t), x$	w^x_t
Witness Holder Algorithms		
WitUp	$\mathsf{stts}, x, w^x_t, \mathsf{upmsg}_{t+1}$	w^x_{t+1}
Verifier or Third Party Algorithms		
VerStatus	$\mathsf{stts}, a_t, x, w^x_t$	$\phi \in \{0,1\}$
Additional Third Party Algorithms in Strong Accumulators		
VerGen	$1^\lambda, S_0, a_0, \mathsf{v}$	$\phi \in \{0,1\}$
VerUpdate	$\mathsf{Op}_t, a_t, a_{t+1}, x, \mathsf{v}_{t+1}$	$\phi \in \{0,1\}$

Fig. 1. Accumulator Algorithms. In static accumulators, the Update, WitUp and VerUpdate algorithms do not exist. In additive accumulators, Op is required to be equal to Add everywhere. In subtractive accumulators, Op is required to be equal to Del. In dynamic accumulators, Op can be either. In positive accumulators, stts is required to be equal to in everywhere. In negative accumulators, stts is required to be equal to out. In universal accumulators, stts can be either.

λ:	The security parameter.
D:	The domain of the accumulator (the set of elements that the accumulator can accumulate). Often, D includes all elements (e.g., $\{0,1\}^*$). Sometimes, D is more limited (e.g., primes of a certain size).
sk:	The accumulator manager's secret key or trapdoor. (The corresponding public key, if one exists, is not modeled here as it can be considered to be a part of the accumulator itself.)
t:	A discrete time / operation counter.
a_t:	The accumulator at time t.
m_t:	Any auxiliary values which might be necessary for the maintenance of the accumulator. These are typically held by the accumulator manager. Note that while the accumulator itself should be constant (or at least sub-linear) in size, m may be larger.
S_t:	The set of elements in the accumulated set at time t. Note that S_0 can be instantiated to be different, based on the initial sets supported by the accumulator in question. Most accumulators assume $S_0 = \emptyset$.
x, y:	Elements which might be added to or removed from the accumulator.
w^x_t:	A witness that element x is (or is not) in the accumulated set at time t.
$\mathsf{stts} \in \{\mathsf{in}, \mathsf{out}\}$:	A flag indicating of whether a given element is in the accumulated set or not.
$\mathsf{Op} \in \{\mathsf{Add}, \mathsf{Del}\}$:	A flag indicating of whether a given element is being added or deleted.
upmsg_t:	A broadcast message sent (by the accumulator manager, if one exists) at time t to all witness holders immediately after the accumulator has been updated. This message is meant to enable all witness holders to update the witnesses they hold for consistency with the new accumulator. It will often contain the new accumulator a_t, and the nature of the update itself (e.g., "x has been added and witness w^x_t has been produced"). It may also contain other information.
v:	A witness that the accumulator a_0 was generated correctly. (Only present in strong accumulators.)
v_t:	A witness that the accumulator a_t was updated correctly. (Only present in strong accumulators.)

Fig. 2. Accumulator Algorithm Input and Output Parameters (from Baldimtsi *et al.* [BCD+17]).

2.2 Security Definitions

A cryptographic accumulator should satisfy two basic security properties: correctness and soundness. In this section, we review the classical correctness and soundness properties of accumulators (stated, for instance, by Ghosh *et al.* [GOP+16]). We revise these classical definitions in several ways.

1. We explicitly consider the correctness of the witness update algorithm, which [GOP+16] consider only as an efficiency shortcut, and thus exclude from their definitions. Since the update algorithm is used in practice, we believe it is important to include in the formal definitions.

2. We allow the generation of membership witnesses during addition (or non-membership witnesses during deletion) as is commonly done in practice, while [GOP$^+$16] only considers the generation of witnesses from a fixed accumulator state. Because of this, we have two separate notions of correctness — *correctness* and *creation-correctness.*

Correctness Definitions. Definitions 1 and 2 give the correctness requirements for the more general case of a universal dynamic accumulator. Informally, an accumulator is *correct* or *creation-correct* if an up-to-date version of a witness produced by Update or $\mathsf{WitCreate}$, respectively, can be used to verify the (non-)membership of the corresponding element. It is easy to adapt our definition for cases of additive/subtractive or positive/negative. To get a definition for an additive accumulator, restrict all instances of Op to be equal to Add; to get a definition for a subtractive accumulator, restrict all instances of Op to be equal to Del. Similarly, to get a definition for a positive accumulator, restrict all instances of stts to be equal to in; to get a definition for a negative accumulator, restrict all instances of stts to be equal to out.

Definition 1 (Correctness). *A universal dynamic accumulator is* correct *for a given domain D of elements if an up-to-date witness w^x corresponding to value x can always be used to verify the (non-)membership of x in an up-to-date accumulator a. More formally, there exists a negligible function ν in the security parameter λ such that for all:*

- *security parameters λ,*
- *initial sets $S_0 \subseteq D$,*
- *values $x \in D$,*
- *positive integers t polynomial in λ,*
- *positive integers t_x such that $1 \leq t_x \leq t$,*
- *operations $\mathsf{Op} \in \{\mathsf{Add}, \mathsf{Del}\}$ (with $\mathsf{stts} = \mathsf{in}$ if $\mathsf{Op} = \mathsf{Add}$ and $\mathsf{stts} = \mathsf{out}$ if $\mathsf{Op} = \mathsf{Del}$),*
- *lists of tuples $[(y_1, \mathsf{Op}_1), \ldots, (y_{t_x-1}, \mathsf{Op}_{t_x-1})]$, $[(y_{t_x+1}, \mathsf{Op}_{t_x+1}), \ldots, (y_t, \mathsf{Op}_t)]$, where*
  - *$y_i \in D$ and $\mathsf{Op}_i \in \{\mathsf{Add}, \mathsf{Del}\}$ for $i \in [1, \ldots, t_x - 1, t_x + 1, \ldots, t]$;*
  - *If $\mathsf{Op} = \mathsf{Add}$, then (x, Del) does not appear in $[(y_{t_x+1}, \mathsf{Op}_{t_x+1}), \ldots, (y_t, \mathsf{Op}_t)]$; and*
  - *If $\mathsf{Op} = \mathsf{Del}$, then (x, Add) does not appear in $[(y_{t_x+1}, \mathsf{Op}_{t_x+1}), \ldots, (y_t, \mathsf{Op}_t)]$,*

The following holds:

$$\Pr\left[\begin{array}{l}(a_0, sk) \leftarrow \mathsf{Gen}(1^\lambda, S_0);\\ (a_i, m_i, w_i^{y_i}, \mathsf{upmsg}_i) \leftarrow \mathsf{Update}(\mathsf{Op}_i, sk, a_{i-1}, m_{i-1}, y_i) \textit{ for } i \in [1, \ldots, t_x - 1];\\ (a_{t_x}, m_{t_x}, w_{t_x}^x, \mathsf{upmsg}_{t_x}) \leftarrow \mathsf{Update}(\mathsf{Op}, sk, a_{t_x-1}, m_{t_x-1}, x);\\ (a_i, m_i, w_i^{y_i}, \mathsf{upmsg}_i) \leftarrow \mathsf{Update}(\mathsf{Op}_i, sk, a_{i-1}, m_{i-1}, y_i) \textit{ for } i \in [t_x + 1, \ldots, t];\\ w_i^x \leftarrow \mathsf{WitUp}(\mathsf{stts}, x, w_{i-1}^x, \mathsf{upmsg}_i) \textit{ for } i \in [t_x + 1, \ldots, t]) :\\ \mathsf{VerStatus}(\mathsf{stts}, a_t, x, w_t^x) = 1\end{array}\right] \geq 1 - \nu(\lambda)$$

Definition 2 (Creation-Correctness). *A universal dynamic accumulator is* creation-correct *for a given domain D of elements if an up-to-date witness w^x* created by the $\mathsf{WitCreate}$ algorithm — not by the Update algorithm! — *corresponding to value x can always be used to verify the (non-)membership of x in an up-to-date accumulator a.*

More formally, there exists a negligible function ν in the security parameter λ such that for all

– *security parameters* λ,
– *initial sets* $S_0 \subseteq D$,
– *values* $x \in D$,
– *positive integers* t *polynomial in* λ,
– *positive integers* t_x *such that* $1 \leq t_x \leq t$,
– *statuses* $\mathsf{stts} \in \{\mathsf{in}, \mathsf{out}\}$, *and*
– *lists of values* $[(y_1, \mathsf{Op}_1), \ldots, (y_t, \mathsf{Op}_t)]$, *where*
  - $y_i \in D$ *and* $\mathsf{Op}_i \in \{\mathsf{Add}, \mathsf{Del}\}$ *for* $i \in [1, \ldots, t]$;
  - *If* $\mathsf{stts} = \mathsf{in}$
    - *either (a)* $x \in S_0$, *or (b)* (x, Add) *appears in* $[(y_1, \mathsf{Op}_1), \ldots, (y_{t_x-1}, \mathsf{Op}_{t_x})]$ *and was not followed by* (x, Del), *and*
    - (x, Del) *does not appear in* $[(y_{t_x+1}, \mathsf{Op}_{t_x+1}), \ldots, (y_t, \mathsf{Op}_t)]$;
  - *If* $\mathsf{stts} = \mathsf{out}$
    - *either (a)* $x \notin S_0$, *or (b)* (x, Del) *appears in* $[(y_1, \mathsf{Op}_1), \ldots, (y_{t_x-1}, \mathsf{Op}_{t_x})]$ *and was not followed by* (x, Add), *and*
    - (x, Add) *does not appear in* $[(y_{t_x+1}, \mathsf{Op}_{t_x+1}), \ldots, (y_t, \mathsf{Op}_t)]$;

The following holds:

$$\Pr\left[\begin{array}{l} (a_0, sk) \leftarrow \mathsf{Gen}(1^\lambda, S_0); \\ (a_i, m_i, w_i^{y_i}, \mathsf{upmsg}_i) \leftarrow \mathsf{Update}(\mathsf{Op}_i, sk, a_{i-1}, m_{i-1}, y_i) \textit{ for } i \in [1, \ldots, t_x]; \\ w_{t_x}^x \leftarrow \mathsf{WitCreate}(\mathsf{stts}, sk, a_t, m_t, x); \\ (a_i, m_i, w_i^{y_i}, \mathsf{upmsg}_i) \leftarrow \mathsf{Update}(\mathsf{Op}_i, sk, a_{i-1}, m_{i-1}, y_i) \textit{ for } i \in [t_x+1, \ldots, t]; \\ w_i^x \leftarrow \mathsf{WitUp}(\mathsf{stts}, x, w_{i-1}^x, \mathsf{upmsg}_i) \textit{ for } i \in [t_x+1, \ldots, t]) : \\ \mathsf{VerStatus}(\mathsf{stts}, a_t, x, w_t^x) = 1 \end{array}\right] \geq 1 - \nu(\lambda)$$

Soundness Definitions. Classically, *collision-freeness* [BP97] is the soundness definition for accumulators. Collision-freeness informally requires that for any element not in the accumulated set it should be hard to find a membership witness. For negative and universal accumulators, collision-freeness can be extended to require that for any element in the accumulated set it should be hard to find a non-membership witness. Another formalization of accumulator soundness for universal accumulators is *undeniability* [Lip12], which requires that for any element (regardless of its presence in the accumulated set) it be hard to find *both* a membership witness and a non-membership witness.

In this paper, we choose to use collision-freeness, since undeniability is not meaningful for positive or negative accumulators, which only support proofs of membership or proofs of non-membership but not both. Definition 3 gives the collision-freeness definition for a universal dynamic accumulator. This definition can be converted to work for positive, negative, additive or subtractive accumulators in the usual way (by limiting the possible values of Op or stts).

Definition 3 (Collision-Freeness). *A universal dynamic accumulator is* collision-free *for a given domain* D *of elements if it is hard to fabricate a (non-)membership witness* w *for a value* x *that is not (or, respectively, is) in the accumulated set. More formally, consider the collision-freeness game described in Figure 3. An accumulator is collision-free if for any sufficiently large security parameter* λ*, for any probabilistic polynomial-time adversary* $\mathcal{A}_{\mathsf{ColFree}}$*, there exists a negligible function* ν *in the security parameter* λ *such that the probability that* $\mathcal{A}_{\mathsf{ColFree}}$ *wins the game is less than* $\nu(\lambda)$.

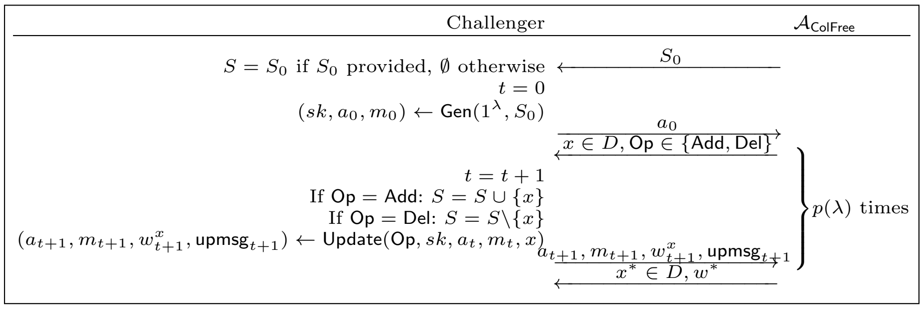
Challenger
$\mathcal{A}_{\mathsf{ColFree}}$
$S = S_0$ if S_0 provided, $\emptyset$ otherwise
S_0
$t = 0$
$(sk, a_0, m_0) \leftarrow \mathsf{Gen}(1^\lambda, S_0)$
a_0
$x \in D, \mathsf{Op} \in \{\mathsf{Add}, \mathsf{Del}\}$
$t = t + 1$
If $\mathsf{Op} = \mathsf{Add}$: $S = S \cup \{x\}$
If $\mathsf{Op} = \mathsf{Del}$: $S = S \backslash \{x\}$
$(a_{t+1}, m_{t+1}, w^x_{t+1}, \mathsf{upmsg}_{t+1}) \leftarrow \mathsf{Update}(\mathsf{Op}, sk, a_t, m_t, x)$
$a_{t+1}, m_{t+1}, w^x_{t+1}, \mathsf{upmsg}_{t+1}$
$p(\lambda)$ times
$x^* \in D, w^*$

Fig. 3. The Collision-Freeness Game. $\mathcal{A}_{\mathsf{ColFree}}$ wins if ($\mathsf{VerStatus}(\mathsf{in}, a_{t+1}, x^*, w^*) = 1$ and $x^* \notin S$), or ($\mathsf{VerStatus}(\mathsf{out}, a_{t+1}, x^*, w^*) = 1$ and $x^* \in S$).

Non-Adaptive Soundness. In the collision-freeness game of Figure 3, the adversary is able to choose elements to add and delete adaptively. However, this notion of collision-freeness (or *soundness*) is quite strong. In a *non-adaptive*[4] version of the game, the adversary would be required to commit to all elements it intends to add before seeing a_0. Certain accumulators can only be shown to meet non-adaptive soundness. One example of such an accumulator is the CLRSAB accumulator, which was informally introduced as a brief remark by Camenisch and Lysyanskaya [CL02] and formally described by Baldimtsi *et al.* [BCD+17]. Note that, in particular, a non-adaptively sound accumulator can always be used to accumulate random values, since it makes no difference whether random values are chosen beforehand or on-the-fly.

Strength. Typically, accumulators are not required to be secure against cheating accumulator managers, since in many scenarios the entity that manages the set (and thus the accumulator) is trusted. When that is not the case (e.g. in many blockchain applications), a *strong* accumulator can be used. A strong accumulator provides guarantees even against a cheating accumulator manager. Informally, an accumulator is *strong* if all of the modifications an accumulator manager makes to the accumulator are verifiable.

Definition 4 (Strength). *An accumulator is* strong *for a given domain D of elements if an adversary cannot win the game described in Figure 3 with non-negligible probability even if it is modified as follows: instead of asking the challenger to run* Gen *and* Update*, the adversary runs them locally and sends the challenger the updated accumulator values together with witnesses* v*. The challenger aborts if* VerGen *or* VerUpdate *return* 0*.*

We must also ensure the correctness of the VerGen and VerUpdate algorithms.

Definition 5 (Strength Correctness). *Informally, an accumulator has* strength correctness *if* VerGen *and* VerUpdate *run on honestly generated inputs and outputs of* Gen *and* Update *always return* 1*.*

[4] Note that this does not refer to non-adaptive corruptions, as in the context of MPC; it is not corruptions that are non-adaptive, but the choice of accumulated elements.

3 Ideal Functionality for Accumulators

Universally Composable (UC) security, proposed by Canetti [Can01] and described briefly in Section 1, requires a different flavor of definitions than those described in Section 2. A UC definition of security for some primitive consists of a set of instructions called an *ideal functionality* which achieves the goals of the primitive when carried out by an incorruptible third party. Informally, to show that a candidate protocol *securely realizes* the ideal functionality, it must be shown that any adversary in a real execution of the protocol can be simulated by a corresponding ideal world adversary in an interaction with the incorruptible third party running the ideal functionality.

Definition 6 ([Can01, Page 12]). *Let $\mathsf{exec}_{\Pi,\mathcal{A},\mathcal{Z}}$ denote the random variable (over the local random choices of all the involved machines) describing the output of environment $\mathcal{Z}$ when interacting with adversary $\mathcal{A}$ and parties running protocol Π. Protocol Π UC-emulates ideal functionality $\mathcal{F}$ if for any adversary $\mathcal{A}$ there exists a simulator $\mathcal{SIM}$ such that, for any environment $\mathcal{Z}$ the distributions of $\mathsf{exec}_{\Pi,\mathcal{A},\mathcal{Z}}$ and $\mathsf{exec}_{\mathcal{F},\mathcal{SIM},\mathcal{Z}}$ are indistinguishable. That is, on any input, the probability that $\mathcal{Z}$ outputs 1 after interacting with $\mathcal{A}$ and parties running Π differs by at most a negligible amount from the probability that $\mathcal{Z}$ outputs 1 after interacting with $\mathcal{SIM}$ and $\mathcal{F}$.*

In this section we present our ideal functionality $\mathcal{F}_{\mathsf{ACC}}$ for an accumulator.

Like [Can04], we discuss several candidate ideal functionalities in order to build up the intuition for how we arrived at the ideal functionality described in Fig. 4 and 5.

First Attempt. A naive first attempt at an accumulator functionality might ignore the accumulator and witness objects altogether, instead functioning as a simple set manager. It would allow the accumulator manager to add and remove elements from the set, and answer 'yes' or 'no' to membership (or non-membership) queries. These queries could optionally be parametrized by timestamps, so as to allow queries about all states of the set, past and present. However, this simple ideal functionality definition fails to support one of the basic modular operations of accumulators. Recall that an accumulator is an object that evolves by time, i.e. at time t it might represent a different set from what it used to represent at time t'. Thus, if we do not consider explicit accumulator objects, then it is impossible to talk about committing to a given set by committing to an accumulator value at a specific time.

Second Attempt - Explicitly Modeling Accumulator Values. A second attempt might be to add explicit accumulator values, without modeling witnesses. So, a membership query would now have the form, 'is this element a (non-)member under this accumulator value?'. However, the absence of explicit witness objects also limits the modular use of accumulators significantly. Specifically, not having explicit witness objects would not work when the ability to verify the (non)membership of certain elements should be secret-shared or otherwise restricted. (For instance, perhaps I should be able to demonstrate my membership in some organization - such as the gym - but any third party shouldn't be able to test my membership without my help, because that would be a violation of my privacy.) Adding these privacy features to an accumulator system would require re-designing and re-proving the accumulator system from scratch if witness objects were not part of the ideal functionality. If witness and accumulator objects are modeled explicitly, however, existing accumulator systems can simply be combined with existing off-the-shelf primitives such as secret sharing, encryption, or commitment. In other words, having the functionality give binary answers to membership queries is over-idealization; it is a good way to model accumulators on their own, but it does not lend itself to use by other protocols that need actual accumulator and witness values to operate.

Final Attempt. Our ideal functionality for accumulators $\mathcal{F}_{\mathsf{ACC}}$ is described in Figures 4 and 5 and provides interfaces for all of the algorithms in Figure 1. (Note that in the functionality the accumulator manager interfaces ignore all queries for which the querier's identity is not encoded in the functionality session id *sid*.)

We loosely base $\mathcal{F}_{\mathsf{ACC}}$ on the ideal functionality for digital signatures described by Canetti [Can04]. Canetti actually gave two different functionalities for digital signatures, which we recall for completeness in Appendix A. The first one (Figure 7) asks the ideal world adversary for a *verification key*; while the second (Figure 8) asks the ideal world adversary for a *verification algorithm.* Similarly, Camenisch *et al.* [CDT19] give functionalities for signatures, non-interactive zero knowledge proofs and for commitments that are explicitly parameterized by the protocol algorithms. Using a given deterministic signature verification algorithm, rather than allowing the ideal world adversary to make each verification decision, achieves two goals:

– It forces verification decisions to be consistent.
– It makes combining UC signatures and zero knowledge proofs of signature knowledge in a black box way simpler.

For these reasons, we chose to define our $\mathcal{F}_{\mathsf{ACC}}$ to receive explicit algorithms from the ideal world adversary. Thus, instead of asking the ideal world adversary to provide updated accumulator states, witnesses and verification decisions, our ideal world adversary provides *all* accumulator algorithms to the functionality (Step 1e in Figure 4).[5] This is a very intuitive way to define an ideal functionality: it explicitly uses the accumulator algorithms except where it needs to modify their behavior to match what is demanded by correctness or soundness. If an ideal execution (that uses the ideal functionality) is indistinguishable from a real execution, that means that the algorithms' behavior did not need any modification.

Just like in the context of digital signatures, if the algorithms are modeled explicitly, usage within multi-party computation (MPC) protocols or in larger zero-knowledge-based systems such as Zcash can be done in a modular way, using existing components.

In addition to the benefits listed above, this also allows us more flexibility to add privacy features to the ideal functionality, as discussed in Section 3.3.

Remark 3. Note that inputs belonging to anyone but the accumulator manager ($\mathcal{AM}$) can be misinformed (just like parties are frequently misinformed about verification keys in signature schemes, in the absence of a PKI). In order to capture such cases, we require parties to provide all inputs to witness holder and third party algorithms, instead of having some inputs, such as the accumulator value, implicitly stored by the ideal functionality.

[5] These algorithms will, among other things, check that elements being added are in the domain D of the accumulator in question.

1. GEN**:** Upon getting (GEN, sid, S_0) as first activation from $\mathcal{AM}$ …
   (a) Initialize an operation counter $t = 0$.
   (b) Initialize an empty list $\mathbf{A}$. This list will be used to keep track of all accumulator states.
   (c) Initialize an empty map $\mathbf{S}$, and set $\mathbf{S}[0] = S_0$. (If S_0 was not provided, use $\emptyset$.) This map will be used to map operation counters to current accumulated sets.
   (d) Send (GEN, sid) to Adversary $\mathcal{A}_{\mathsf{Ideal}}$.
   (e) Get $(\mathsf{ALGORITHMS}, sid, (\mathsf{Gen}, \mathsf{Update}, \mathsf{WitCreate}, \mathsf{WitUp}, \mathsf{VerStatus}, \mathsf{VerGen}, \mathsf{VerUpdate}))$ from Adversary $\mathcal{A}_{\mathsf{Ideal}}$. This includes all of the accumulator algorithms; their expected input output behavior is described in Figure 1. All of them should be polynomial-time; we restrict the verification algorithms $\mathsf{VerStatus}, \mathsf{VerGen}, \mathsf{VerUpdate}$ to be deterministic.
   (f) Run $(sk, a_0, m_0, \mathsf{v}) \leftarrow \mathsf{Gen}(1^\lambda, S_0)$.
   (g) Verify that $\mathsf{VerGen}(S_0, a_0, \mathsf{v}) = 1$. If not, output $\perp$ to $\mathcal{AM}$ and halt. (This ensures strength.) Otherwise, continue.
   (h) Store sk, m_0; add a_0 to $\mathbf{A}$.
   (i) Output $(\mathsf{ALGORITHMS}, sid, S_0, (\mathsf{Gen}, \mathsf{Update}, \mathsf{WitCreate}, \mathsf{WitUp}, \mathsf{VerStatus}, \mathsf{VerGen}, \mathsf{VerUpdate})$ to $\mathcal{AM}$.
2. UPDATE**:** Upon getting $(\mathsf{UPDATE}, sid, \mathsf{Op}, x)$ from $\mathcal{AM}$ …
   (a) Increment the operation counter: $t = t + 1$.
   (b) Set $\mathbf{S}[t] = \mathbf{S}[t-1]$.
   (c) Run $(a_t, m_t, w_t^x, \mathsf{upmsg}_t, \mathsf{v}_t) \leftarrow \mathsf{Update}(\mathsf{Op}, sk, a_{t-1}, m_{t-1}, x)$.
   (d) If $\mathsf{Op} = \mathsf{Add}$:
      i. Verify that $\mathsf{VerStatus}(\mathsf{in}, a, x, w_t) = 1$. If not, output $\perp$ to $\mathcal{AM}$ and halt. (This ensures correctness.) Otherwise, continue.
      ii. If $x \notin \mathbf{S}[t]$, add x to $\mathbf{S}[t]$.
   (e) If $\mathsf{Op} = \mathsf{Del}$:
      i. Verify that $\mathsf{VerStatus}(\mathsf{out}, a, x, w_t) = 1$. If not, output $\perp$ to $\mathcal{AM}$ and halt. (This ensures negative correctness.) Otherwise, continue.
      ii. If $x \in \mathbf{S}[t]$, remove x from $\mathbf{S}[t]$.
   (f) Verify that $\mathsf{VerUpdate}(\mathsf{Op}, a_{t-1}, a_t, x, \mathsf{v}_t) = 1$. If not, output $\perp$ to $\mathcal{AM}$ and halt. (This ensures strength.) Otherwise, continue.
   (g) Store m_t, upmsg_t; add a_t to $\mathbf{A}$.
   (h) Output $(\mathsf{UPDATE}, sid, \mathsf{Op}, a_t, x, w_t, \mathsf{upmsg}_t)$ to $\mathcal{AM}$.
3. $\mathsf{WITCREATE}$**:** Upon getting $(\mathsf{WITCREATE}, sid, \mathsf{stts}, x)$ from $\mathcal{AM}$ …
   (a) Run $w \leftarrow \mathsf{WitCreate}(\mathsf{stts}, sk, a_t, m_t, x, (\mathsf{upmsg}_1, \ldots, \mathsf{upmsg}_t))$
   (b) If $\mathsf{stts} = \mathsf{in}$:
   If $x \in \mathbf{S}[t]$, verify that $\mathsf{VerStatus}(\mathsf{in}, a_t, x, w) = 1$. If not, output $\perp$ to $\mathcal{AM}$ and halt. (This ensures creation-correctness.) Otherwise, continue.
   (c) If $\mathsf{stts} = \mathsf{out}$:
   If $x \notin \mathbf{S}[t]$, verify that $\mathsf{VerStatus}(\mathsf{out}, a_t, x, w) = 1$. If not, output $\perp$ to $\mathcal{AM}$ and halt. (This ensures negative-creation-correctness.) Otherwise, continue.
   (d) Output $(\mathsf{WITNESS}, sid, \mathsf{stts}, x, w)$ to $\mathcal{AM}$.
4. WITUP**:** Upon getting $(\mathsf{WITUP}, sid, \mathsf{stts}, a_{old}, a_{new}, x, w_{old}, (\mathsf{upmsg}_{old+1}, \ldots, \mathsf{upmsg}_{new}))$ from any party $\mathcal{H}$ …
   (a) Run $w_{new} \leftarrow \mathsf{WitUp}(\mathsf{stts}, x, w_{old}, (\mathsf{upmsg}_{old+1}, \ldots, \mathsf{upmsg}_{new}))$
   (b) If $a_{old} \in \mathbf{A}$, $a_{new} \in \mathbf{A}$ and $old < new$:
      i. If $\mathsf{stts} = \mathsf{in}$, $\mathsf{VerStatus}(\mathsf{in}, a_{old}, x, w_{old}) = 1$, $x \in \mathbf{S}[t]$ for $t \in [old, \ldots, new]$, $\mathsf{upmsg}_{old+1}, \ldots, \mathsf{upmsg}_{new}$ match the stored values and $\mathsf{VerStatus}(\mathsf{in}, a_{new}, x, w_{new}) = 0$, output $\perp$ to $\mathcal{P}$ and halt. (This ensures correctness.) Otherwise, continue.
      ii. If $\mathsf{stts} = \mathsf{out}$, $\mathsf{VerStatus}(\mathsf{out}, a_{old}, x, w_{old}) = 1$, $x \notin \mathbf{S}[t]$ for $t \in [old, \ldots, new]$, $\mathsf{upmsg}_{old+1}, \ldots, \mathsf{upmsg}_{new}$ match the stored values and $\mathsf{VerStatus}(\mathsf{out}, a_{new}, x, w_{new}) = 0$, output $\perp$ to $\mathcal{P}$ and halt. (This ensures negative correctness.) Otherwise, continue.
   (c) Output $(\mathsf{UPDATEDWITNESS}, sid, \mathsf{stts}, a_{old}, a_{new}, x, w_{old}, (\mathsf{upmsg}_{old+1}, \ldots, \mathsf{upmsg}_{new}), w_{new})$ to $\mathcal{H}$.

Fig. 4. Ideal Functionality $\mathcal{F}_{\mathsf{ACC}}$ for Accumulators With Explicit Verification Algorithm

The ideal functionality described in Figures 4 and 5 is really an entire "menu" of functionalities covering all different types of accumulators: additive, subtractive, dynamic, positive, negative and universal and finally strong accumulators. More explicitly, by default, if all of the text (except for the text colored by pink) is considered, the ideal functionality describes a dynamic, universal accumulator. By restricting Op to be only Add or only Del we could make it additive or subtractive instead of dynamic; by restricting stts to be only in or only out we could make it positive or negative instead of universal. Figure 4 describes the ideal functionality interfaces for the accumulator manager and witness holders; Figure 5 describes the interfaces for third parties.

```
1. VERSTATUS: Upon getting $(\mathsf{VERSTATUS}, sid, \mathsf{stts}, a, \mathsf{VerStatus}', x, w)$ from any party $\mathcal{P}$ ...
   (a) If $\mathsf{VerStatus}' = \mathsf{VerStatus}$ and there exists a $t$ such that $a = a_t \in \mathbf{A}$:
       i. Let $t$ be the largest such number.
       ii. If $\mathsf{stts} = \mathsf{in}$:
           A. If $\mathcal{AM}$ not corrupted, $x \notin \mathbf{S}[t]$ and $\mathsf{VerStatus}(\mathsf{in}, a_t, x, w) = 1$, output $\perp$ to $\mathcal{P}$ and halt.
              (This ensures collision-freeness.) Otherwise, continue.
           B. Set $\phi = \mathsf{VerStatus}(\mathsf{in}, a_t, x, w)$.
       iii. If $\mathsf{stts} = \mathsf{out}$:
           A. If $\mathcal{AM}$ not corrupted, $x \in \mathbf{S}[t]$ and $\mathsf{VerStatus}(\mathsf{out}, a_t, x, w) = 1$, output $\perp$ to $\mathcal{P}$ and
              halt. (This ensures negative collision-freeness.) Otherwise, continue.
           B. Set $\phi = \mathsf{VerStatus}(\mathsf{out}, a_t, x, w)$.
   (b) Otherwise, set $\phi = \mathsf{VerStatus}'(\mathsf{stts}, a, x, w)$.
   (c) Output $(\mathsf{VERIFIED}, sid, \mathsf{stts}, a, \mathsf{VerStatus}', x, w, \phi)$ to $\mathcal{P}$.
2. VERGEN: Upon getting $(\mathsf{VERGEN}, sid, S, a, \mathsf{v}, \mathsf{VerGen}')$ from any party $\mathcal{P}$ ...
   (a) Set $\phi = \mathsf{VerGen}'(S, a, \mathsf{v})$.
   (b) Output $(\mathsf{VERIFIED}, sid, S, a, \mathsf{v}, \mathsf{VerGen}', \phi)$ to $\mathcal{P}$.
3. VERUPDATE: Upon getting $(\mathsf{VERUPDATE}, sid, \mathsf{Op}, a, a', x, \mathsf{v}_t, \mathsf{VerUpdate}')$ from any party $\mathcal{P}$ ...
   (a) Set $\phi = \mathsf{VerUpdate}'(\mathsf{Op}, a, a', x, \mathsf{v}_t)$.
   (b) Output $(\mathsf{VERIFIED}, sid, \mathsf{Op}, a, a', x, \mathsf{v}_t, \mathsf{VerUpdate}', \phi)$ to $\mathcal{P}$.
```

Fig. 5. Ideal Functionality $\mathcal{F}_{\mathsf{ACC}}$ Interfaces for Third Parties

We use color coding to describe different types of accumulators within the same functionality. If the ideal functionality is limited to the black text, it describes a positive additive accumulator. Actions that are present only in subtractive accumulators are colored green. Actions that are present only in negative accumulators are colored blue. Finally, actions that are present only in strong accumulators are colored pink; actions *not* present in strong accumulators are colored orange.

We use $\mathcal{F}_{\mathsf{ACC}}$ to refer to the universal dynamic accumulator functionality. We add Add, Del, in and out to the subscript to denote additive, subtractive, positive and negative accumulators, respectively. We add other parameters to the subscript (e.g. 'STRONG') to denote other properties.

3.1 Modeling Decentralized Management

If the accumulator is strong, it may make sense to allow anyone to perform an accumulator update, instead of restricting the ability to perform such updates to the accumulator manager. We model this by making a few changes to the functionality. First, the GEN, UPDATE and WITCREATE interfaces of the ideal functionality no longer only accept invocations by $\mathcal{AM}$. Additionally, instead of having a strict ordering of update operations, we might allow parties to perform an update on any accumulator state, resulting in a tree of states. The functionality will be modified to perform the appropriate checks and record-keeping.

3.2 Modeling Non-Adaptive Soundness

We model non-adaptive soundness (Section 2.2) by making two simple changes to the ideal functionality. First, when sending the GEN command to the ideal functionality (in Step 1 of Figure 4), the accumulator manager $\mathcal{AM}$ is expected to provide a set of all elements that will ever be added or deleted. (This can be done e.g. by providing a PRF seed.) Second, if even one element outside of that set is added or deleted, nothing is guaranteed; the functionality simply runs the algorithms it was given, without performing any checks.

3.3 Adding Privacy Properties

Our ideal functionality as stated in Figures 4 and 5 does not make any attempt to hide anything about the accumulated set from any accumulator user. In this section, we discuss how we add such privacy properties to the ideal functionality.

Add-Delete Unlinkability. In certain scenarios it is desirable that an adversary should not be able to link an addition of an element to a deletion of the same element later on. Such a property is relevant when accumulators are used as an anonymous revocation mechanism where the revocation information should not allow anyone to determine that the user revoked just now was the user who joined two hours ago, and not the user who joined four hours ago [BCD+17]. We do not formally model add-delete unlinkability; instead, we define a stronger property which we call *hiding update-message (HUM).*

Hiding Update-Message (HUM). Informally, an accumulator is *hiding update-message*, or *HUM*, if given all of the update messages produced in the course of an execution, it is impossible to tell whether one specific update message corresponds to the addition / deletion of element x_0 or element x_1 for $x_0, x_1 \in D$.

We can incorporate HUM into our ideal functionality by placing limitations on the algorithm Update provided by the ideal world adversary. We require Update to consist of two sub-algorithms: one sub-algorithm — Update_1 — which receives no input at all except for randomness, and produces the update message; and a second sub-algorithm — Update_2 — which can receive state from Update_1 as well as all of the other inputs typically provided to Update, and produces all the other outputs of Update. This forces update messages to reveal nothing about the added / deleted element.

Note that this modification is very strong, since it forces the update messages to *statistically* hide the elements; constructions where the elements are only computationally hidden would not meet this definition. This modification trivially implies the add-delete unlinkability property described above, since update messages now contain no information at all about the elements.

Remark 4. We clearly need to withhold x from Update_1, in order to guarantee that the update message does not reveal x. However, we could consider allowing Update_1 to see the other inputs to Update. This would not work because if we give Update_1 access to the accumulator a or the auxiliary value m, then the update message it produces might contain arbitrary information about the set of elements accumulated prior to the current operation. In particular, the update message might reveal which elements were added / deleted previously, breaking the HUM property.

Zero-Knowledge. Ghosh *et al.* [GOP+16] define the notion of a *zero-knowledge accumulator*, which requires that accumulator and witness values reveal nothing about the accumulated set (other than the element to which the witness corresponds). We can incorporate ZK by placing limitations on the Update and $\mathsf{WitCreate}$ algorithms provided by the ideal world adversary, just like we did for the HUM property. We can require each algorithm to consist of two sub-algorithms: one which does not require any set-dependent inputs and produces the accumulator and witness values (as necessary), and a second sub-algorithm (which can receive state from the first) which produces all other values.

3.4 Discussion: Incorrect Accumulator and Witness Values

If an incorrect accumulator value (or verification algorithm $\mathsf{VerStatus}'$) is provided to the verification interface, we allow the party making the query to control the verification verdict, via $\mathsf{VerStatus}'$. This models the fact that any party can issue verification queries for accumulator values of their choice — for instance, for accumulator values which they may have generated themselves, and for which they control the accumulated set.

If an incorrect witness for a member element is provided to the verification interface, we allow the ideal world adversary to control the verification verdict (via the algorithm $\mathsf{VerStatus}$ it provides during the generation phase). This models the fact that we only require the ideal world adversary to be unable to come up with a witness for a *non*-member (or a non-membership witness for a member); we do not require that an adversary be unable to come up with a witness for a member (or a non-membership witness for a non-member). For instance, it may be possible to modify valid witnesses to obtain other witnesses for the same element. Note also that multiple witnesses can be generated for the same element by means of the $\mathsf{WitCreate}$ interface.

4 Equivalence Argument

Like Canetti [Can04], we prove that satisfying our UC definition for dynamic universal accumulators is the same as satisfying the classical definition.[6]

Theorem 1. *Let* $\Pi_{\mathsf{ACC}} = (\mathsf{Gen}, \mathsf{Update}, \mathsf{WitCreate}, \mathsf{WitUp}, \mathsf{VerStatus})$ *be a universal dynamic accumulator scheme, and let* $\mathsf{VerStatus}$ *be deterministic. Then* Π_{ACC} *securely realizes* $\mathcal{F}_{\mathsf{ACC}}$ *if and only if* Π_{ACC} *satisfies Definitions 1, 2 and 3.*

Proof. Our proof follows the structure of the proof of Canetti [Can04] (pages 12-14).

1. We start by assuming that Π_{ACC} does not satisfy Definitions 1, 2 and 3. We then show that Π_{ACC} also does not securely realize $\mathcal{F}_{\mathsf{ACC}}$. To do this, we build an environment $\mathcal{Z}$ and an adversary $\mathcal{A}_{\mathsf{Real}}$ such that for any simulator $\mathcal{SIM}$, $\mathcal{Z}$ can distinguish between interacting with $\mathcal{A}_{\mathsf{Real}}$ and Π_{ACC}, and interacting with $\mathcal{SIM}$ and $\mathcal{F}_{\mathsf{ACC}}$. Like the environment of Canetti [Can04], our environment does not corrupt any parties, and does not send any messages to the adversary. Because all accumulator operations are non-interactive, meaning that they are run locally by individual parties, no messages are exchanged in the real world. So, the adversary $\mathcal{A}_{\mathsf{Real}}$ is never activated.
   (a) Assume Π_{ACC} is not correct (i.e. does not satisfy Definition 1). That is, there exists a security parameter λ, an initial set $S_0 \subseteq D$, a value $x \in D$, an operation $\mathsf{Op} \in \{\mathsf{Add}, \mathsf{Del}\}$ (with $\mathsf{stts} = \mathsf{in}$ if $\mathsf{Op} = \mathsf{Add}$ and $\mathsf{stts} = \mathsf{out}$ if $\mathsf{Op} = \mathsf{Del}$) and a list of values $[(y_1, \mathsf{Op}_1), \dots, (y_{t_x-1}, \mathsf{Op}_{t_x-1})]$, $[(y_{t_x+1}, \mathsf{Op}_{t_x+1}), \dots, (y_t, \mathsf{Op}_t)]$, where
      – $y_i \in D$ and $\mathsf{Op}_i \in \{\mathsf{Add}, \mathsf{Del}\}$ for $i \in [1, \dots, t_x - 1, t_x + 1, \dots, t]$;
      – If $\mathsf{Op} = \mathsf{Add}$, then (x, Del) does not appear in $[(y_{t_x+1}, \mathsf{Op}_{t_x+1}), \dots, (y_t, \mathsf{Op}_t)]$; and
      – If $\mathsf{Op} = \mathsf{Del}$, then (x, Add) does not appear in $[(y_{t_x+1}, \mathsf{Op}_{t_x+1}), \dots, (y_t, \mathsf{Op}_t)]$,

[6] This proof also implies that satisfying our UC definition for additive or subtractive, positive or negative accumulators is the same as satisfying the classical definition; however, it does not imply anything for strong accumulators. We leave that up to future work.

such that with non-negligible probability, the honestly-produced witness for x against accumulator a_t will not verify.

Our environment $\mathcal{Z}$ will send the following commands to some party $\mathcal{AM}$, where sid encodes the identity of $\mathcal{AM}$:

– (GEN, sid, S_0),
– $(\mathsf{UPDATE}, sid, \mathsf{Op}_1, y_1), \ldots, (\mathsf{UPDATE}, sid, \mathsf{Op}_{t_x-1}, y_{t_x-1})$,
– $(\mathsf{UPDATE}, sid, \mathsf{Op}, x)$, and
– $(\mathsf{UPDATE}, sid, \mathsf{Op}_{t_x+1}, y_{t_x+1}), \ldots, (\mathsf{UPDATE}, sid, \mathsf{Op}_t, y_t)$.

As a result of the third step, $\mathcal{Z}$ will learn a_{t_x} and $w^x_{t_x}$. As a result of the fourth step, $\mathcal{Z}$ will learn a_t and $t - t_x$ update messages $(\mathsf{upmsg}_{t_x+1}, \ldots, \mathsf{upmsg}_t)$. It then sends $(\mathsf{WITUP}, \mathsf{stts}, sid, a_{t_x}, a_t, x, w^x_{t_x}, (\mathsf{upmsg}_{t_x+1}, \ldots, \mathsf{upmsg}_t))$ to some party $\mathcal{H}$ (where possibly $\mathcal{H} = \mathcal{AM}$), and receives w^x_t back. Finally, it sends $(\mathsf{VERSTATUS}, sid, \mathsf{stts}, a_t, \mathsf{VerStatus}' = \mathsf{VerStatus}(\cdot,\cdot,\cdot,\cdot), x, w^x_t)$ to some party $\mathcal{P}$ (which may be the same party or not). $\mathcal{Z}$ outputs the returned verdict ϕ.

In the real world, ϕ will be 0 with non-negligible probability according to our assumption.

In the ideal world, if no error messages are returned, ϕ will always be 1, since in WitUp, we will always hit Item 4(b)i or 4(b)ii of Figure 4, and there the first three listed conditions will be satisfied.

(b) Assume Π_{ACC} is not creation-correct (i.e. does not satisfy Definition 2). $\mathcal{Z}$ can distinguish between the real and ideal worlds in a way very similar to that described above.

(c) Assume Π_{ACC} is not collision-free (i.e. does not satisfy Definition 3). That is, there exists an adversary $\mathcal{A}_{\mathsf{ColFree}}$ that can forge a (non-)membership witness for a non-member (or member, respectively) x with non-negligible probability. Our $\mathcal{Z}$ will use $\mathcal{A}_{\mathsf{ColFree}}$ to generate inputs for $\mathcal{AM}$. Having received x^*, w^* from $\mathcal{A}_{\mathsf{ColFree}}$, $\mathcal{Z}$ will compute ϕ_{in} by calling $(\mathsf{VERSTATUS}, sid, \mathsf{in}, a_t, x^*, w^*)$, and ϕ_{out} by calling $(\mathsf{VERSTATUS}, sid, \mathsf{out}, a_t, x^*, w^*)$. $\mathcal{Z}$ will then output 1 if x^* was in the accumulated set and $\phi_{out} = 1$ or if x^* was not in the accumulated set and $\phi_{in} = 1$, and will output 0 otherwise.

In the real world, if $\mathcal{A}_{\mathsf{ColFree}}$ met the collision-freeness win conditions, $\mathcal{Z}$ will output 1 with non-negligible probability according to our assumption.

In the ideal world, both ϕ_{in} and ϕ_{out} will always be 0 or $\perp$, since we will satisfy the first two conditions in Item 1(a)iiA (or Item 1(a)iiiA, if $\mathsf{stts} = \mathsf{out}$) of $\mathsf{VERSTATUS}$ in Figure 5. If the third condition is satisfied too, $\perp$ will be returned. If it is not, 0 will be returned, as a result of Item 1(a)iiB (or Item 1(a)iiiB, if $\mathsf{stts} = \mathsf{out}$) in Figure 5.

2. We now prove the other direction. Assume that Π_{ACC} does not securely realize $\mathcal{F}_{\mathsf{ACC}}$. That is, there exists an adversary $\mathcal{A}_{\mathsf{Real}}$ such that for any simulator $\mathcal{SIM}$, there exists an environment $\mathcal{Z}$ that can distinguish between interacting with $\mathcal{A}_{\mathsf{Real}}$ and Π_{ACC}, and interacting with $\mathcal{SIM}$ and $\mathcal{F}_{\mathsf{ACC}}$. We show that if that is the case, Π_{ACC} must also violate Definition 1, 2 or 3. We pick a simulator $\mathcal{SIM}$ that proceeds as follows, running an internal copy of $\mathcal{A}_{\mathsf{Real}}$:

– Inputs from $\mathcal{Z}$ is forwarded to $\mathcal{A}_{\mathsf{Real}}$. Outputs from $\mathcal{A}_{\mathsf{Real}}$ is forwarded to $\mathcal{Z}$.
– $\mathcal{SIM}$ handles corruptions according to the standard corruption model [Can01].
– Upon receiving (GEN, sid) from $\mathcal{F}_{\mathsf{ACC}}$, $\mathcal{SIM}$ sends the actual accumulator algorithms back as $(\mathsf{GEN}, sid, (\mathsf{Gen}, \mathsf{Update}, \mathsf{WitCreate}, \mathsf{WitUp}, \mathsf{VerStatus}))$.

This simulator guarantees that the real and ideal worlds will be distributed identically, *unless* one of the following causes $\mathcal{F}_{\mathsf{ACC}}$ to return $\perp$:

- In Update, $\mathcal{F}_{\mathsf{ACC}}$ hits Item 2(d)i or 2(e)i of Figure 4. If this happens, correctness (Definition 1) is violated.
- In $\mathsf{WitCreate}$, $\mathcal{F}_{\mathsf{ACC}}$ hits Item 3b or 3c of Figure 4. If this happens, creation-correctness (Definition 2) is violated.
- In $\mathsf{VerStatus}$, $\mathcal{F}_{\mathsf{ACC}}$ hits Item 1(a)iiA or 1(a)iiiA of Figure 5. If this happens, collision-freeness (Definition 3) is violated.
- In WitUp, $\mathcal{F}_{\mathsf{ACC}}$ hits Item 4(b)i or 4(b)ii of Figure 4. If this happens, either correctness or creation-correctness is violated.

In order for $\mathcal{Z}$ to distinguish between the real and ideal worlds, one of the above must happen with non-negligible probability, and thus either Definition 1, 2 or 3 must be violated with non-negligible probability.

We can modify the theorem and proof to also prove equivalence between classical and UC definitions for strong accumulators.

Corollary 1. *Let* $\Pi_{\mathsf{ACC}} = (\mathsf{Gen}, \mathsf{Update}, \mathsf{WitCreate}, \mathsf{WitUp}, \mathsf{VerStatus}, \mathsf{VerGen}, \mathsf{VerUpdate})$ *be a strong universal dynamic accumulator scheme, and let* $\mathsf{VerStatus}$, VerGen *and* $\mathsf{VerUpdate}$ *be deterministic. Then* Π_{ACC} *securely realizes* $\mathcal{F}_{\mathsf{ACC},STRONG}$ *if and only if* Π_{ACC} *satisfies Definitions 1, 2 and 4.*

Proof. The proof is very similar to that of Theorem 1 above, with a few changes. The changes are in Steps 1c and 2 of the proof.

In Step 1c of the proof above, instead of calling $\mathcal{A}_{\mathsf{ColFree}}$, we call $\mathcal{A}_{\mathsf{Strength}}$ which runs Gen and Update itself. The environment $\mathcal{Z}$ computes its output exactly as before. In the ideal world, both ϕ_{in} and ϕ_{out} will always be 0 or $\perp$, since we will satisfy the first condition in Item 1(a)iiA (or Item 1(a)iiiA, if $\mathsf{stts} = \mathsf{out}$) of $\mathsf{VERSTATUS}$ (ignoring the condition that $\mathcal{AM}$ is not corrupted, which does not apply for a strong accumulator). If the third condition is satisfied too, $\perp$ will be returned. If it is not, 0 will be returned, as a result of Item 1(a)iiB (or 1(a)iiiB, if $\mathsf{stts} = \mathsf{out}$) of Figure 5.

In Step 2 of the proof above, $\mathcal{SIM}$ includes VerGen and $\mathsf{VerUpdate}$ in the list of algorithms it sends to the ideal functionality. Then, in the list of things that might cause $\mathcal{F}_{\mathsf{ACC},STRONG}$ to return $\perp$, we replace the third bullet with the following:

- In $\mathsf{VerStatus}$, $\mathcal{F}_{\mathsf{ACC}}$ hits Item 1(a)iiA or 1(a)iiiA of Figure 5. If this happens, strength (Definition 4) is violated.

We also add the following:

- In Gen, $\mathcal{F}_{\mathsf{ACC},STRONG}$ returns $\perp$ at Item 1g of Figure 4. If this happens, strength correctness (Definition 5) is violated.
- In Update, $\mathcal{F}_{\mathsf{ACC},STRONG}$ returns $\perp$ at Item 2f of Figure 4. If this happens, strength correctness (Definition 5) is violated.

5 Demonstrations of Composition

We now present two examples of accumulator composition to showcase the convenience of having UC secure accumulators.

5.1 Accumulator Composition: Braavos

Baldimtsi *et al.* [BCD$^+$17] show how one can build accumulators with certain properties by composing other types of (potentially weaker) accumulators. Among other examples, Baldimtsi *et al.* build the *Braavos* accumulator. We present a modified version of Braavos, which we call Braavos$'$. Braavos$'$ is a *hiding update-message (HUM)* dynamic accumulator as described in Section 3.3. We describe Braavos$'$ in Figure 6.

Just like Braavos, Braavos$'$ leverages the following two weaker accumulators:

1. SIG: A positive (but not dynamic) additive accumulator, in the form of a digital signature scheme. Note that this accumulator does not have any update messages (and we thus omit update messages from its inputs and outputs).
2. CLRSAB: A non-adaptively-sound negative additive accumulator. One example of such an accumulator is the CLRSAB construction[7], informally introduced as a brief remark by Camenisch and Lysyanskaya [CL02] and formally described by Baldimtsi *et al.* [BCD$^+$17].

Informally, Braavos$'$ works as follows. When a new element x is added, a random value r_x is chosen to correspond to it, and the pair (x, r_x) is accumulated in SIG. Since we use a digital signature scheme, no update message is sent. A proof of membership for x consists of the value r_x, a proof of membership of (x, r_x) in SIG (which is simply a digital signature), and a proof of non-membership of r_x in CLRSAB. Then, when the element x is deleted, r_x is added to CLRSAB (so a proof of non-membership of r_x in CLRSAB can no longer be produced). Next time x is added, a fresh random value is chosen, and so forth.

Unlike Braavos$'$, Braavos [BCD$^+$17] uses the same random value every time a given element is re-added, instead of choosing fresh random values. This has the advantage of saving on accumulator manager storage requirements. However, it has the disadvantage that deletions of the same element can all be linked to one another, since the same random value is present in all of the associated update messages. This violates the HUM property[8] (but not the add-delete unlinkability property, which is the one Baldimtsi *et al.* require).

Braavos$'$ is obviously HUM, since it (a) has empty update messages for additions, and (b) has update messages for deletions that are completely independent of the element being deleted. Intuitively, it is secure because if an element was never added then no signature on it exists, and every time an element x is removed, all random values r_x that have been signed with x are in the CLRSAB accumulator, so no proof of non-membership for any such r_x can be produced.

More formally, let $\mathcal{F}_{\mathsf{ACC},\mathsf{in},HUM}$ be our accumulator functionality $\mathcal{F}_{\mathsf{ACC}}$ for a dynamic, positive, HUM accumulator. That is, $\mathcal{F}_{\mathsf{ACC},\mathsf{in},HUM}$ is $\mathcal{F}_{\mathsf{ACC}}$ restricted to $\mathsf{stts} = \mathsf{in}$, and requiring the simulator to provide Update in two parts, as necessary for HUM (described in Section 3.3). Similarly, let $\mathcal{F}_{\mathsf{ACC},\mathsf{in},\mathsf{Add}}$ be our accumulator functionality $\mathcal{F}_{\mathsf{ACC}}$ for a

[7] The CLRSAB accumulator is actually universal and dynamic, but we only require it to be negative and additive.

[8] Adding zero knowledge proofs would not resolve this issue — that random value cannot be hidden within a zero knowledge proof in any straightforward way, since it must be used to update CLRSAB witnesses.

positive additive accumulator, and let $\mathcal{F}_{\mathsf{ACC},\mathsf{out},\mathsf{Add},NA}$ be our accumulator functionality $\mathcal{F}_{\mathsf{ACC}}$ for a negative additive accumulator that is non-adaptively sound (Section 3.2).

Theorem 2. *The Braavos′ accumulator described in Figure 6 securely realizes* $\mathcal{F}_{\mathsf{ACC},\mathsf{in},HUM}$ *as long as* SIG *securely realizes* $\mathcal{F}_{\mathsf{ACC},\mathsf{in},\mathsf{Add}}$ *with no update messages, and* CLRSAB *securely realizes* $\mathcal{F}_{\mathsf{ACC},\mathsf{out},\mathsf{Add},NA}$.

We can prove Theorem 2 very simply using the fact that both SIG and CLRSAB are UC-secure (that is, by operating in the double-$\mathcal{F}_{\mathsf{ACC}}$-hybrid model). Before our UC definitions, a proof of security would involve a multi-step security reduction of the new accumulator to one of the old ones.

Proof. The simulator for the new accumulator uses its two inner simulators to obtain algorithms for the inner accumulators, composes them as in Figure 6, and submits those to the ideal functionality. (Since the CLRSAB accumulator is only non-adaptively sound, the simulator also pre-selects the random values that are to be accumulated in the CLRSAB accumulator.)[9]

In Section 3.3, we described how in order to modify the UC functionality $\mathcal{F}_{\mathsf{ACC}}$ to be HUM, we require that the simulator provide the algorithm Update in two parts: one sub-algorithm (let's call it Update_1) which only receives randomness and produces the update message; and a second sub-algorithm (let's call it Update_2) which produces all the other outputs of Update, and is additionally allowed to depend on the state of Update_1. If the update being performed is an addition, we do not need Update_1 at all, since no update message is necessary; we simply set $\mathsf{Update}_2(\mathsf{Add}, sk, a_t, m_t, x) = \mathsf{Update}(\mathsf{Add}, sk, a_t, m_t, x)$. If the update being performed is a deletion, $\mathsf{Update}_1(\mathsf{Del}, sk, a_t, m_t)$ gets a random pre-selected value and performs a CLRSAB addition on it; it then passes the random value it added as $\mathsf{state}_{\mathsf{Update}_1}$ to $\mathsf{Update}_2(\mathsf{Del}, sk, a_t, m_t, x, \mathsf{state}_{\mathsf{Update}_1})$ which does the rest of the work.

The views of the environment $\mathcal{Z}$ in the real and ideal worlds will be identical in the so-called double-$\mathcal{F}_{\mathsf{ACC}}$-hybrid model, since the sub-accumulator functionalities guarantee that if an element was never added then no signature on it exists, and every time an element x is removed, all random values r_x that have been signed with x are in set accumulated in CLRSAB, so no proof of non-membership for any such r_x can be produced.

5.2 Accumulators for Anonymous Credentials

We now informally discuss how our UC definition of accumulators would simplify the security proof of a complex system like anonymous credentials. An ideal functionality that provides all the properties of anonymous credentials including pseudonyms, selective attribute disclosure, predicates over attributes, revocation, inspection, etc. is described by Camenisch *et al.* [CDHK15]. (Baldimtsi *et al.* [BCD$^+$17] augment this functionality with revocation.) In this section, to demonstrate the benefits of modularity we concentrate on a simplified version of an anonymous credential ideal functionality with three types of parties: the credential manager or issuer, credential holders, and credential verifiers. Our ideal functionality has the following interfaces for the credential manager:

[9] Notice that this works regardless of how the simpler accumulators are implemented (simply software vs. hardward vs. distributed protocols), since they satisfy the UC definition.

```
$\mathsf{Gen}(1^\lambda, S)$:
    1. $(\mathsf{SIG}.sk, \mathsf{SIG}.a_0) \leftarrow \mathsf{SIG}.\mathsf{Gen}(1^\lambda, \emptyset)$
    2. $(\mathsf{CLRSAB}.sk, \mathsf{CLRSAB}.a_0, \mathsf{CLRSAB}.\mathsf{upmsg}_0) \leftarrow \mathsf{CLRSAB}.\mathsf{Gen}(1^\lambda, \emptyset)$
    3. Set
        (a) $sk \leftarrow (\mathsf{SIG}.sk, \mathsf{CLRSAB}.sk)$,
        (b) $a_0 \leftarrow (\mathsf{SIG}.a_0, \mathsf{CLRSAB}.a_0)$,
        (c) $\mathsf{upmsg}_0 \leftarrow \mathsf{CLRSAB}.a_0$
        (d) Instantiate $m_0$ as an empty map.
    4. Return $(sk, a_0, \mathsf{upmsg}_0, m_0)$
$\mathsf{Update}(\mathsf{Op}_t, sk, a_t, m_t, x)$:
    1. If $\mathsf{Op}_t = \mathsf{Add}$ and $x \notin m_t$:
        (a) Pick $r_x$ at random from the domain $D_{\mathsf{CLRSAB}}$ of the CLRSAB accumulator. (We require
            the domain to be large enough that the probability of picking the same element twice is
            negligible.)
        (b) Set $m_{t+1} = m_t$
        (c) Set $m_{t+1}[x] = r_x$
        (d) $\mathsf{CLRSAB}.w_{t+1}^{r_x} \leftarrow \mathsf{CLRSAB}.\mathsf{WitCreate}(\mathsf{out}, \mathsf{CLRSAB}.sk, \mathsf{CLRSAB}.a_t, r_x)$
        (e) $\mathsf{SIG}.w_{t+1}^{(x,r_x)} \leftarrow \mathsf{SIG}.\mathsf{Update}(\mathsf{Add}, \mathsf{SIG}.sk, \mathsf{SIG}.a_0, (x, r_x))$
        (f) Set $\mathsf{CLRSAB}.a_{t+1} = \mathsf{CLRSAB}.a_t$.
        (g) Set $a_{t+1} = (\mathsf{SIG}.a_0, \mathsf{CLRSAB}.a_{t+1})$
        (h) Set $w_{t+1}^x = (r_x, \mathsf{CLRSAB}.w_{t+1}^{r_x}, \mathsf{SIG}.w_{t+1}^{(x,r_x)})$
        (i) Set $\mathsf{upmsg}_{t+1} = \perp$
        (j) Return $(a_{t+1}, m_{t+1}, w_{t+1}^x, \mathsf{upmsg}_{t+1})$
    2. If $\mathsf{Op}_t = \mathsf{Del}$ and $x \in m_t$:
        (a) Set $r_x = m_t[x]$
        (b) Set $m_{t+1} = m_t$
        (c) Delete $x$ from $m_{t+1}$
        (d) $(\mathsf{CLRSAB}.a_{t+1}, \mathsf{CLRSAB}.\mathsf{upmsg}_{t+1}) \leftarrow \mathsf{CLRSAB}.\mathsf{Update}(\mathsf{Add}, \mathsf{CLRSAB}.sk, \mathsf{CLRSAB}.a_t, r_x)$
        (e) Set $a_{t+1} = (\mathsf{SIG}.a_0, \mathsf{CLRSAB}.a_{t+1})$
        (f) Set $\mathsf{upmsg}_{t+1} = \mathsf{CLRSAB}.\mathsf{upmsg}_{t+1}$
        (g) Return $(a_{t+1}, m_{t+1}, \mathsf{upmsg}_{t+1})$
$\mathsf{WitCreate}(\mathsf{stts}, sk, a_t, m_t, x)$:
    1. If $\mathsf{stts} = \mathsf{in}$ and $x \in m_t$:
        (a) Set $r_x = m_t[x]$
        (b) $\mathsf{SIG}.w_t^{(x,r_x)} \leftarrow \mathsf{SIG}.\mathsf{WitCreate}(\mathsf{in}, \mathsf{SIG}.sk, \mathsf{SIG}.a_t, (x, r_x))$
        (c) $\mathsf{CLRSAB}.w_t^{r_x} \leftarrow \mathsf{CLRSAB}.\mathsf{WitCreate}(\mathsf{out}, \mathsf{CLRSAB}.sk, \mathsf{CLRSAB}.a_t, r_x)$
        (d) Set $w_t^x = (r_x, \mathsf{CLRSAB}.w_t^{r_x}, \mathsf{SIG}.w_t^{(x,r_x)})$
        (e) Return $w_t^x$
$\mathsf{WitUp}(\mathsf{stts}, x, w_t^x = (r_x, \mathsf{CLRSAB}.w_t^{r_x}, \mathsf{SIG}.w_t^{(x,r_x)}), \mathsf{upmsg}_{t+1})$:
    1. If $\mathsf{upmsg}_{t+1} \neq \perp$: (This update message corresponds to a deletion)
        (a) $\mathsf{CLRSAB}.w_{t+1}^{r_x} = \mathsf{CLRSAB}.\mathsf{WitUp}(\mathsf{out}, r_x, \mathsf{CLRSAB}.w_t^{r_x}, \mathsf{upmsg}_{t+1})$
    2. Else: $w_{t+1}^x = w_t^x$
    3. Return $w_{t+1}^x$
$\mathsf{VerStatus}(\mathsf{in}, a_t = (\mathsf{SIG}.a_t, \mathsf{CLRSAB}.a_t), x, w_t^x = (r_x, \mathsf{CLRSAB}.w_t^{r_x}, \mathsf{SIG}.w_t^{(x,r_x)}))$:
    1. Return 1 if both of the following are 1, and 0 otherwise:
        – $\mathsf{SIG}.\mathsf{VerStatus}(\mathsf{in}, \mathsf{SIG}.a_t, (x, r_x), \mathsf{SIG}.w_t^{(x,r_x)})$
        – $\mathsf{CLRSAB}.\mathsf{VerStatus}(\mathsf{out}, \mathsf{CLRSAB}.a_t, r_x, \mathsf{CLRSAB}.w_t^{r_x})$
```

Fig. 6. Braavos′ Algorithms. We omit parameters unnecessary for the SIG and CLRSAB accumulator algorithms.

1. KeyGen, to set up the scheme parameters.
2. IssueCred(*token*, *property*), to issue a credential certifying *property* to a credential holder who knows the secret corresponding to *token*, and
3. RevokeCred(*token*, *property*), to revoke an issued credential.

Our simplified functionality sends the simulator all information about issued and revoked credentials (including *token* and *property* information); so, unlike the full-fledged functionality of Camenisch *et al.*, it does not restrict access to information about who is certified for what property.

Credential holders only have a single interface — ProveCred, which they use to demonstrate to a credential verifier that they hold a credential certifying some property. Credential holders should be able to use their credentials anonymously. The credential verifiers have the corresponding interface VerifyCredProof, which allows them to check the proof provided by the credential holder.

Now, imagine that we instantiate our simplified anonymous credential functionality with a combination of the following building blocks: (a) digital signatures, (b) accumulators and (c) (non-interactive) zero knowledge (ZK) proofs, as described by Baldimtsi *et al.* [BCD$^+$17]. A simple instantiation would work as follows:

The signatures are used simply to guarantee the authenticity of updates made by the credential manager. KeyGen sets up the parameters for all three primitives. IssueCred adds $(token, property)$ to the accumulator, where $token$ is a value linked to a long-term secret belonging to the user (e.g. $token$ might be a public key), and $property$ is the property the credential certifies (e.g. "citizen", "member", "age = 30", etc.). Similarly, RevokeCred deletes the appropriate element from the accumulator. Whenever an update happens to the accumulator value, the most recent value (and a corresponding update message) is signed by the credential manager and sent to all system users, who can then bring their accumulator witnesses up to date.

ProveCred would then provide a ZK proof of knowledge of long-term user secret s, token $token$ and accumulator witness w such that, for the most recent credential-manager-signed accumulator, the conjunction of the following statements is true:

1. s is appropriately linked to $token$ (through some relationship, e.g. s is the secret key corresponding to $token$ which is the public key), and
2. the accumulator verification algorithm returns true when given the accumulator witness w and $(token, property)$.

Given that no UC accumulator existed before our work, in order for someone to prove security even of such a simple scheme, a reduction would be required that would reduce the security of the overall scheme to the underlying building blocks. However, we can prove the security of this simplified credential scheme in the UC model using UC secure versions of the underlying building blocks. Such a UC proof would be information theoretic and unconditional, and will hold for any implementation of the underlying primitives, whether they be simple software, distributed computation, hardware, etc.

In order to prove the security of this credential scheme we need to build a simulator that, when run with the ideal functionality, produces an environment view indistinguishable from that of a real run of the anonymous credentials protocol. The two difficulties in doing so is (1) playing the roles of honest parties without knowing their long-term secrets, and (2) arguing that real adversaries can no more convince verifiers to accept forged credentials than ideal functionality adversaries can. UC zero knowledge proofs address the first concern. Since the use of UC zero knowledge proofs allows the simulator to control the zero knowledge proof ideal functionality (which we review in Appendix B), it can control the verification outcome without actually knowing the values in question, sidestepping this issue. UC accumulators address the second concern.

Acknowledgements

This research was supported, in part, by US NSF grant 1717067.

References

[ACJT00] Giuseppe Ateniese, Jan Camenisch, Marc Joye, and Gene Tsudik. A practical and provably secure coalition-resistant group signature scheme. In Mihir Bellare, editor, *CRYPTO 2000*, volume 1880 of *LNCS*, pages 255–270. Springer, Heidelberg, August 2000.

[BCD$^+$17] Foteini Baldimtsi, Jan Camenisch, Maria Dubovitskaya, Anna Lysyanskaya, Leonid Reyzin, Kai Samelin, and Sophia Yakoubov. Accumulators with applications to anonymity-preserving revocation. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*, pages 301–315. IEEE, 2017.

[Bd94] Josh Cohen Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital sinatures (extended abstract). In Tor Helleseth, editor, *EUROCRYPT'93*, volume 765 of *LNCS*, pages 274–285. Springer, Heidelberg, May 1994.

[BP97] Niko Bari and Birgit Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 480–494. Springer, Heidelberg, May 1997.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[Can04] Ran Canetti. Universally composable signature, certification, and authentication. In *Proceedings of the 17th IEEE Workshop on Computer Security Foundations*, CSFW '04, pages 219–, Washington, DC, USA, 2004. IEEE Computer Society.

[CDHK15] Jan Camenisch, Maria Dubovitskaya, Kristiyan Haralambiev, and Markulf Kohlweiss. Composable and modular anonymous credentials: Definitions and practical constructions. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part II*, volume 9453 of *LNCS*, pages 262–288. Springer, Heidelberg, November / December 2015.

[CDT19] Jan Camenisch, Manu Drijvers, and Björn Tackmann. Multi-protocol UC and its use for building modular and efficient protocols. *IACR Cryptology ePrint Archive*, 2019:65, 2019.

[CF01] Ran Canetti and Marc Fischlin. Universally composable commitments. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 19–40. Springer, Heidelberg, August 2001.

[CHKO08] Philippe Camacho, Alejandro Hevia, Marcos A. Kiwi, and Roberto Opazo. Strong accumulators from collision-resistant hashing. In Tzong-Chen Wu, Chin-Laung Lei, Vincent Rijmen, and Der-Tsai Lee, editors, *ISC 2008*, volume 5222 of *LNCS*, pages 471–486. Springer, Heidelberg, September 2008.

[CKS09] Jan Camenisch, Markulf Kohlweiss, and Claudio Soriente. An accumulator based on bilinear maps and efficient revocation for anonymous credentials. In Stanislaw Jarecki and Gene Tsudik, editors, *PKC 2009*, volume 5443 of *LNCS*, pages 481–500. Springer, Heidelberg, March 2009.

[CKS11] Jan Camenisch, Stephan Krenn, and Victor Shoup. A framework for practical universally composable zero-knowledge protocols. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 449–467. Springer, Heidelberg, December 2011.

[CL02] Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 61–76. Springer, Heidelberg, August 2002.

[CV02] Jan Camenisch and Els Van Herreweghen. Design and implementation of the idemix anonymous credential system. In Vijayalakshmi Atluri, editor, *ACM CCS 2002*, pages 21–30. ACM Press, November 2002.

[GOP$^+$16] Esha Ghosh, Olga Ohrimenko, Dimitrios Papadopoulos, Roberto Tamassia, and Nikos Triandopoulos. Zero-knowledge accumulators and set algebra. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part II*, volume 10032 of *LNCS*, pages 67–100. Springer, Heidelberg, December 2016.

[Lip12] Helger Lipmaa. Secure accumulators from euclidean rings without trusted setup. In Feng Bao, Pierangela Samarati, and Jianying Zhou, editors, *ACNS 12*, volume 7341 of *LNCS*, pages 224–240. Springer, Heidelberg, June 2012.

[LLX07] Jiangtao Li, Ninghui Li, and Rui Xue. Universal accumulators with efficient nonmembership proofs. In Jonathan Katz and Moti Yung, editors, *ACNS 07*, volume 4521 of *LNCS*, pages 253–269. Springer, Heidelberg, June 2007.

[MGGR13] Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. Zerocoin: Anonymous distributed E-cash from Bitcoin. In *2013 IEEE Symposium on Security and Privacy*, pages 397–411. IEEE Computer Society Press, May 2013.

[Ngu05] Lan Nguyen. Accumulators from bilinear pairings and applications. In Alfred Menezes, editor, *CT-RSA 2005*, volume 3376 of *LNCS*, pages 275–292. Springer, Heidelberg, February 2005.

```
1. Key Generation: Upon getting (KEYGEN, $sid$) from a party Signer ...
   (a) If this is not the first KeyGen command, ignore this command. Otherwise, continue.
   (b) If $sid$ does not encode Signer's identity, ignore this command. Otherwise, continue.
   (c) Initialize an empty map $\mathbf{W}$.
   (d) Send (KEYGEN, $sid$) to Adversary $\mathcal{A}_{\mathsf{Ideal}}$.
   (e) Get (VERKEY, $sid, vk$) from Adversary $\mathcal{A}_{\mathsf{Ideal}}$.
   (f) Record $vk$.
   (g) Send (VERKEY, $sid, vk$) to Signer.
2. Signature Generation: Upon getting (SIGN, $sid, x$) from a party Signer ...
   (a) Verify that $sid$ encodes Signer's identity. If not, ignore this command. Otherwise, continue.
   (b) Send (SIGN, $sid, x$) to Adversary $\mathcal{A}_{\mathsf{Ideal}}$.
   (c) Get (SIGNATURE, $sid, x, \sigma$) from Adversary $\mathcal{A}_{\mathsf{Ideal}}$.
   (d) Verify that $(x, \sigma) \notin \mathbf{W}$ or $\mathbf{W}[(x, \sigma)] = 1$. If not, send $\perp$ to Signer and halt. Otherwise, continue.
   (e) If $(x, \sigma) \notin \mathbf{W}$, record $\mathbf{W}[(x, \sigma)] = 1$.
   (f) Output (SIGNATURE, $sid, x, \sigma$) to Signer.
3. Signature Verification: Upon getting (VERIFY, $sid, x, \sigma, vk$) from a party Verifier ...
   (a) Send (VERIFY, $sid, x, \sigma, vk$) to Adversary $\mathcal{A}_{\mathsf{Ideal}}$.
   (b) Get (VERIFIED, $sid, x, \sigma, vk, \phi$) from Adversary $\mathcal{A}_{\mathsf{Ideal}}$.
   (c) If $(x, \sigma) \in \mathbf{W}$: let $\phi' = \mathbf{W}[(x, \sigma)]$.
   (d) Else:
       i. If the signer is not corrupted, $vk$ is the recorded public key, and $(x, \sigma) \notin \mathbf{W}$, set $\phi' = 0$.
       ii. Else, let $\phi' = \phi$.
       iii. Record $\mathbf{W}[(x, \sigma)] = \phi'$.
   (e) Output (VERIFIED, $sid, x, \sigma, vk, \phi'$) to Verifier.
```

Fig. 7. Ideal Functionality for Digital Signatures [Can04]

A Universally Composable Signatures

In this appendix (specifically, in Figures 7 and 8), we describe the two digital signature ideal functionalities described by Canetti [Can01,Can04]. The first does not require the simulator to provide the signing and verification algorithms explicitly at key generation time; the second does. Both ideal functionalities require the verifier to provide the verification key (or verification algorithm) when using the verification interface. This models the fact that the verifier might be misinformed about the verification key if a PKI is not available.

B Universally Composable Zero-Knowledge

In this appendix (in Figure 9) we recall the ideal functionality $\mathcal{F}_{ZK}$ from [Can01] which is parameterized by a binary relation R that takes in an element x and a witness w. It expects a single input (PROVE, sid, x, w) from Prover (where sid encodes the identities of Prover and Verifier). If $R(x, w) = 1$ then $\mathcal{F}_{ZK}$ will output (VERIFIED, sid, x) to Verifier[10].

C The RSA Accumulator

In this appendix (in Figures 10, 11 and 12), we review the RSA dynamic universal accumulator, which has been shown to meet the classical accumulator definitions. By Theorem 1, it follows that this accumulator also meets our UC definition.

The RSA accumulator is described across several papers. It was introduced by Benaloh and de Mare [Bd94], augmented with dynamism by Camenisch and Lysyanskaya [CL02], and with universality by Li, Li and Xue [LLX07].

[10] Corruption is also modeled; if Prover is corrupt, the adversary learns the prover's witness w.

1. **Key Generation:** Upon getting (KEYGEN, sid) from a party Signer ...
   (a) If this is not the first KeyGen command, ignore this command. Otherwise, continue.
   (b) If sid does not encode Signer's identity, ignore this command. Otherwise, continue.
   (c) Initialize an empty list $\mathbf{W}$ of signed messages.
   (d) Send (KEYGEN, sid) to Adversary $\mathcal{A}_{\mathsf{Ideal}}$.
   (e) Get ($\mathsf{ALGORITHMS}, sid, \mathsf{Sign}, \mathsf{Ver}$) from Adversary $\mathcal{A}_{\mathsf{Ideal}}$, where Sign is a polynomial-time algorithm and Ver is a polynomial-time *deterministic* algorithm.
   (f) Send ($\mathsf{ALGORITHMS}, sid, \mathsf{Ver}$) to Signer.
2. **Signature Generation:** Upon getting (SIGN, sid, x) from a party Signer ...
   (a) Verify that sid encodes Signer's identity. If not, ignore this command. Otherwise, continue.
   (b) Let $\sigma = \mathsf{Sign}(x)$.
   (c) Verify that $\mathsf{Ver}(x, \sigma) = 1$. If not, send $\perp$ to Signer and halt. Otherwise, continue.
   (d) Output ($\mathsf{SIGNATURE}, sid, x, \sigma$) to Signer.
   (e) Record x in $\mathbf{W}$.
3. **Signature Verification:** Upon getting ($\mathsf{VERIFY}, sid, x, \sigma, \mathsf{Ver}'$) from a party $\mathsf{Verifier}$ ...
   (a) If $\mathsf{Ver}' = \mathsf{Ver}$, the signer is not corrupted, $\mathsf{Ver}(x, \sigma) = 1$ and $x \notin \mathbf{W}$, send $\perp$ to signer and halt. (This violates soundness.) Otherwise, continue.
   (b) $\phi = \mathsf{Ver}'(x, \sigma)$.
   (c) Output ($\mathsf{VERIFIED}, sid, x, \sigma, \mathsf{Ver}', \phi$) to $\mathsf{Verifier}$.

Fig. 8. Ideal Functionality for Digital Signatures With Explicit Verification Algorithm [Can01] (2005 version)

$\mathcal{F}_{ZK}^{R}$ is parameterized by a binary relation R. It proceeds as follows.

1. Upon getting ($\mathsf{PROVE}, sid, x, w$) from Prover, Ignore it unless $sid = (\mathsf{Prover}, \mathsf{Verifier}, sid')$ for some $\mathsf{Verifier}$. Next, if $R(x, w) = 1$, send the output ($\mathsf{VERIFIED}, sid, x$) to $\mathsf{Verifier}$; otherwise, do nothing. From now on, ignore PROVE inputs.
2. Upon getting ($\mathsf{CORRUPTPROVER}, sid$) from Adversary $\mathcal{A}_{\mathsf{Ideal}}$, send w to Adversary $\mathcal{A}_{\mathsf{Ideal}}$. If Adversary $\mathcal{A}_{\mathsf{Ideal}}$ now provides a value (x', w') such that $R(x', w')$ holds, and no output was yet sent to $\mathsf{Verifier}$, send ($\mathsf{VERIFIED}, sid, x'$) to $\mathsf{Verifier}$.

Fig. 9. Ideal Functionality for Zero Knowledge [Can01] (2005 version)

```
$\mathsf{Gen}(1^\lambda, S_0)$:
    1. Select two $\lambda$-bit safe primes $p = 2p' + 1$ and $q = 2q' + 1$ where $p'$ and $q'$ are also prime, and
       let $n = pq$. (Consider $n$ to be public knowledge from hereon out; it is actually a part of the
       accumulator value $a$, but for simplicity we will not refer to it as such.)
    2. Let $sk = p'q'$.
    3. Select a random integer $a' \leftarrow \mathbb{Z}_n^*$.
    4. Let $a_0 = (a')^2 \bmod n$. (Like $n$, this initial accumulator $a_0$ will be part of all future accumulator
       values, but for simplicity we will not refer to it as such.)
    5. Let $m_0 = S_0$ be the set of member elements.
    6. Return $(sk, a_0, m_0)$.
$\mathsf{Update}(\mathsf{Op}, sk, a_t, m_t, x)$:
    1. Let $m_{t+1} = m_t$.
    2. Check that $x \in D$ (that is, that $x$ is an odd prime). If not, return $\perp$.
    3. If $\mathsf{Op} = \mathsf{Add}$:
        (a) If $x \in m_{t+1}$ ($x$ is already a member):
                i. Let $w_{t+1}^x = \mathsf{WitCreate}(\mathsf{in}, sk, a_t, m_t, x)$.
               ii. Let $a_{t+1} = a_t$.
              iii. Let $\mathsf{upmsg}_t = (\mathsf{Add}, \perp)$.
        (b) Else ($x$ is not yet a member):
                i. Add $x$ to $m_{t+1}$.
               ii. Let $w_{t+1}^x = a_t$.
              iii. Let $a_{t+1} = a_t^x \bmod n$.
               iv. Let $\mathsf{upmsg}_{t+1} = (\mathsf{Add}, x)$.
        (c) Return $(a_{t+1}, m_{t+1}, w_{t+1}^x, \mathsf{upmsg}_{t+1})$.
    4. If $\mathsf{Op} = \mathsf{Del}$:
        (a) If $x \notin m_t$ ($x$ is already a non-member):
                i. Let $a_{t+1} = a_t$.
               ii. Let $\mathsf{upmsg}_{t+1} = (\mathsf{Del}, \perp)$.
        (b) Else ($x$ is a member):
                i. Remove $x$ from $m_{t+1}$.
               ii. Let $a_{t+1} = a_t^{x^{-1} \bmod sk} \bmod n$.
              iii. Let $\mathsf{upmsg}_{t+1} = (\mathsf{Del}, (a_{t+1}, x))$.
        (c) Let $w_{t+1}^x = \mathsf{WitCreate}(\mathsf{out}, sk, a_{t+1}, m_{t+1}, x)$.
        (d) Return $(a_{t+1}, m_{t+1}, w_{t+1}^x, \mathsf{upmsg}_{t+1})$.
$\mathsf{WitCreate}(\mathsf{stts}, sk, a_t, m_t, x)$:
    1. If $x \notin D$ or $x \notin m_t$, $\mathsf{FAIL}$.
    2. If $\mathsf{stts} = \mathsf{in}$:
        (a) Let $w_t^x = a_t^{x^{-1} \bmod p'q'} \bmod n$, and return $w_t^x$.
    3. If $\mathsf{stts} = \mathsf{out}$:
        (a) Compute Bezout coefficients $\alpha$, $\beta$ such that $\alpha(\prod_{y \in m_{t+1}} y) + \beta x = 1$. (Given that $x \neq y$
            for all $y \in m_{t+1}$, since both $x$ and all $y$ are prime, such $\alpha$ and $\beta$ are guaranteed to exist
            and be efficiently computable.)
        (b) Let $r_t^x = \alpha$, and $s_t^x = a_0^{-\beta} \bmod n$.
        (c) Return $w_t^x = (r_t^x, s_t^x)$.
       Verifying that the witness holds, we have

       $$a_t^{r_t^x} = a_0^{r_t^x \prod_{y \in m_{t+1}} y} = a_0^{\alpha \prod_{y \in m_{t+1}} y} = a_0^{1 - \beta x} = a_0 (a_0^{-\beta})^x = a_0 (s_t^x)^x \pmod{n}.$$
```

Fig. 10. RSA Accumulator Manager Algorithms

```
$\mathsf{WitUp}(\mathsf{stts}, x, w_t^x, \mathsf{upmsg}_{t+1})$:
    1. Parse $(\mathsf{Op}, \mathsf{upmsg}') = \mathsf{upmsg}$.
    2. If $\mathsf{stts} = \mathsf{in}$:
        (a) If $\mathsf{Op} = \mathsf{Add}$:
            i. Parse $y = \mathsf{upmsg}'$.
            ii. If $y = \perp$: let $w_{t+1}^x = w_t^x$.
            iii. Else: let $w_{t+1}^x = (w_t^x)^y \bmod n$.
            iv. Return $w_{t+1}^x$.
        (b) If $\mathsf{Op} = \mathsf{Del}$:
            i. If $\mathsf{upmsg}' = \perp$: let $w_{t+1}^x = w_t^x$.
            ii. Else:
                A. Parse $(a_{t+1}, y) = \mathsf{upmsg}'$.
                B. Compute Bezout coefficients $\alpha$, $\beta$ such that $\alpha x + \beta y = 1$. (Given that $x \neq y$, since
                   both $x$ and $y$ are prime, such $\alpha$ and $\beta$ are guaranteed to exist.)
                C. Let $w_{t+1}^x = (w_t^x)^\beta a_{t+1}^\alpha \bmod n$.
            iii. Return $w_{t+1}^x$.
    3. If $\mathsf{stts} = \mathsf{out}$:
        (a) Parse $(r_t^x, s_t^x) = w_t^x$.
        (b) If $\mathsf{Op} = \mathsf{Add}$:
            i. Parse $y = \mathsf{upmsg}'$.
            ii. If $y = \perp$: let $w_{t+1}^x = w_t^x$.
            iii. Else:
                A. Find $\alpha$ and $\beta$ such that $\alpha y + \beta x = 1$.
                B. $r_{t+1}^x = \alpha r_t^x \bmod x$.
                C. Find a value $v$ such that $r_{t+1}^x y = r_t^x + vx$.
                D. $s_{t+1}^x = s_t^x a_t^v \bmod n$.
                E. Let $w_{t+1}^x = (r_{t+1}^x, s_{t+1}^x)$.
                F. Return $w_{t+1}^x$.
                Verifying that the new witness holds, we have

                    $$a_{t+1}^{r_{t+1}} = a_t^{r_{t+1}y} = a_t^{r_t + rx} = a_t^{rx} a_t^{r_t}$$

                    $$= a_t^{rx} s^x a_0 = (a_t^r s)^x a_0 = s_{t+1}^x a_0 (\mathrm{mod} n).$$

        (c) $\mathsf{Op} = \mathsf{Del}$:
            i. If $\mathsf{upmsg}' = \perp$: let $w_{t+1}^x = w_t^x$.
            ii. Else:
                A. Find a value $v$ such that $0 \leq r_t^x y - vx < 2^\lambda$ (this is efficiently doable).
                B. $r_{t+1}^x = r_t^x y - vx$.
                C. $s_{t+1}^x = s_t^x a_{t+1}^x a_0$.
                D. Let $w_{t+1}^x = (r_{t+1}^x, s_{t+1}^x)$.
                E. Return $w_{t+1}^x$.
                Verifying that the new witness holds, we have

                    $$a_{t+1}^{r_{t+1}} = a_{t+1}^{r_t y - rx} = s_t^x a_0 a_{t+1}^{-rx} = (s_t a_{t+1}^{-1})^x a_0 = s_{t+1}^x a_0 (\mathrm{mod} n).$$
```

Fig. 11. RSA Witness Holder Algorithms

```
$\mathsf{VerStatus}(\mathsf{stts}, a_t, x, w_t^x)$:
    1. If $\mathsf{stts} = \mathsf{in}$:
        (a) Return 1 if $a_t = (w_t^x)^x \bmod n$.
        (b) Return 0 otherwise.
    2. If $\mathsf{stts} = \mathsf{out}$:
        (a) Parse $(r_t^x, s_t^x) = w_t^x$.
        (b) Return 1 if $a_t^{r_t^x} = (s_t^x)^x a_0 (\mathrm{mod} n)$.
        (c) Return 0 otherwise.
```

Fig. 12. RSA Verifier Algorithms