On Signatures of Knowledge

Melissa Chase and Anna Lysyanskaya

Computer Science Department
Brown University
Providence, RI 02912
{mchase,anna}@cs.brown.edu

Abstract. In a traditional signature scheme, a signature σ on a message m is issued under a public key PK, and can be interpreted as follows: "The owner of the public key PK and its corresponding secret key has signed message m." In this paper we consider schemes that allow one to issue signatures on behalf of any NP statement, that can be interpreted as follows: "A person in possession of a witness w to the statement that $x \in L$ has signed message m." We refer to such schemes as *signatures of knowledge.*

We formally define the notion of a signature of knowledge. We begin by extending the traditional definition of digital signature schemes, captured by Canetti's ideal signing functionality, to the case of signatures of knowledge. We then give an alternative definition in terms of games that also seems to capture the necessary properties one may expect from a signature of knowledge. We then gain additional confidence in our two definitions by proving them equivalent.

We construct signatures of knowledge under standard complexity assumptions in the common-random-string model.

We then extend our definition to allow signatures of knowledge to be *nested* i.e., a signature of knowledge (or another accepting input to a UC-realizable ideal functionality) can itself serve as a witness for another signature of knowledge. Thus, as a corollary, we obtain the first *delegatable* anonymous credential system, i.e., a system in which one can use one's anonymous credentials as a secret key for issuing anonymous credentials to others.

1 Introduction

Digital signature schemes constitute a cryptographic primitive of central importance. In a traditional digital signature scheme, there are three algorithms: (1) the key generation algorithm KeyGen through which a signer sets up his public and secret keys; (2) the signing algorithm Sign; and (3) the verification algorithm Verify. A signature in a traditional signature scheme can be thought of as an assertion *on behalf of a particular public key.* One way to interpret (m, σ) where $\mathsf{Verify}(PK, m, \sigma) = Accept$, is as follows: "the person who generated public key PK and its corresponding secret key SK has signed message m."

We ask ourselves the following question: Can we have a signature scheme in which a signer can speak *on behalf of any NP statement to which he knows a* *witness*? For example, let ϕ be a Boolean formula. Then we want anyone who knows a satisfying assignment w to be able to issue tuples of the form (m, σ), where $\mathsf{Verify}(\phi, m, \sigma) = \mathit{Accept}$, that can be interpreted as follows: "a person who knows a satisfying assignment to formula ϕ has signed message m." Further, we ask whether we can have a signature that just reveals that statement but nothing else; in particular, it reveals nothing about the witness. Finally, what if we want to use a signature issued this way as a witness for issuing another signature?

Online, you are what you know, and access to data is what empowers a user to authenticate her outgoing messages. The question is: *what* data? Previously, it was believed that a user needed a public signing key associated with her identity, and knowledge of the corresponding secret key is what gave her the power to sign. Surprisingly, existence of signatures of knowledge means that if there is *any* NP statement $x \in L$ is associated with a user's identity, the knowledge of a corresponding and hard-to-find witness w for this statement is sufficient to empower the user to sign.

Why We Need Signatures of Knowledge as a New Primitive. Suppose that a message m is signed under some public key PK, and σ is the resulting signature. This alone is not sufficient for any application to trust the message m, unless this application has reason to trust the public key PK. Thus, in addition to (m, σ, PK), such an application will also request some proof that PK is trustworthy, e.g., a certification chain rooted at some trusted PK_0. In order to convince others to accept her signature, the owner of the public key PK has to reveal a lot of information about herself, namely, her entire certification chain. Yet, all she was trying to communicate was that the message m comes from someone trusted by the owner of PK_0. Indeed, this is all the information that the application needs to accept the message m. If instead the user could issue a *signature of knowledge* of her SK, PK, and the entire certification chain, she would accomplish the same goal without revealing all the irrelevant information.

More generally, for any polynomial-time Turing machine M_L, we want to be able to sign using knowledge of a witness w such that $M_L(x, w) = \mathit{Accept}$. We think of M_L as a procedure that decides whether w is a valid witness for $x \in L$ for the NP language L. We call the resulting signature *a signature of knowledge of w that is a witness to $x \in L$, on message m*, or sometimes just a signature of knowledge of w on message m, or sometimes a signature of knowledge on behalf of $x \in L$ on message m.

Other Applications Our simplest example is a ring signature [20]. In a ring signature, a signer wishes to sign a message m in such a way that the signature cannot be traced to her specifically, but instead to a group of N potential signers, chosen at signing time. A ring signature can be realized by issuing a signature of knowledge of one of the secret keys corresponding to N public keys. Moreover, following Dodis et al. [16] using cryptographic accumulators [4], the size of this ring signature need not be proportional to N: simply accumulate all public keys into one accumulator A using a public accumulation function, and then issue a signature of knowledge of a secret key corresponding to a public key in A.

Next, let us show how signatures of knowledge give rise to a simple group signature scheme [11, 7, 1, 2, 5]. In a group signature scheme, we have group members, a group manager, and an anonymity revocation manager. Each member can sign on behalf of the group, and a signature reveals no information about who signed it, unless the anonymity revocation manager gets involved. The anonymity revocation manager can trace the signature to the group member who issued it; moreover it is impossible, even if the group manager and the revocation manager collude, to create a signature that will be traced to a group member who did not issue it.

Consider the following simple construction. The group's public key consists of (PK_s, PK_E, f), where PK_s is a signature verification key for which the group manager knows the corresponding secret key; PK_E is an encryption public key for which the anonymity revocation manager knows the corresponding decryption key; and f is a one-way function. To become a group member, a user picks a secret x, gives $f(x)$ to the group manager and obtains a group membership certificate $g = \sigma_{PK_s}(f(x))$. To issue a group signature, the user picks a random string R, encrypts his identity using randomness R: $c = \mathit{Enc}(PK_E, f(x); R)$ and produces a signature of knowledge σ of (x, g, R) such that c is an encryption of $f(x)$ using randomness R, and g is a signature on $f(x)$. The resulting group signature consists of (c, σ). To trace a group signature, the revocation manager decrypts c. It is not hard to see (only intuitively, since we haven't given any formal definitions) that this construction is a group signature scheme. Indeed, at a high level, this is how existing practical and provably secure group signatures work [1, 5].

Unlike the two applications above that have already been studied and where signatures of knowledge offer just a conceptual simplification, our last application was not known to be realizable prior to this work.

Consider the problem of delegatable anonymous credentials. The problem can be explained using the following example. Suppose that, as Brown University employees, we have credentials attesting to that fact, and we can use these credentials to open doors to campus facilities. We wish to be able do so anonymously because we do not want the janitors to monitor our individual whereabouts. Now suppose that we have guests visiting us. We want to be able to issue them a guest pass using our existing credential as a secret key, and without revealing any additional information about ourselves, even to our guests. In turn, our visitors should be able to use their guest passes in order to issue credentials to their taxi drivers, so these drivers can be allowed to drive on the Brown campus. So we have a credential delegation chain, from the Brown University certification authority (CA) that issues us the employee credential, to us, to our visitors, to the visitors' taxi drivers, and each participant in the chain does not know who gave him/her the credential, but (1) knows the length of his credential chain and knows that this credential chain is rooted at the Brown CA; and (2) can extend the chain and issue a credential to the next person.

Although it may seem obvious how to solve this problem once we cast everything in terms of signatures of knowledge and show how to realize signatures of knowledge, we must stress that this fact eluded researchers for a very long time, dating back to Chaum's original vision of the world with anonymous credentials [10]. More recently this problem was raised in the anonymous credentials literature [19, 6, 18]. And it is still elusive when it comes to practical protocols: our solution is not efficient enough to be used in practice.

On Defining Signatures of Knowledge. The first definition of any new primitive is an attempt to formalize intuition. We see from the history of cryptographic definitions (from defining security for encryption, signatures, multiparty computation) that it requires a lot of effort and care. Our approach is to give two definitions, each capturing our intuition in its own way, and then prove that they are equivalent.

One definitional approach is to give an ideal functionality that captures our intuition for a signature of knowledge. Our ideal functionality will guarantee that a signature will only be accepted if the functionality sees the witness w either when generating the signature or when verifying it; and, moreover, signatures issued by signers through this functionality will always be accepted. At the same time, the signatures that our functionality will generate will contain no information about the witness. This seems to capture the intuitive properties we require of a signature of knowledge, although there are additional subtleties we will discuss in Section 2.1. For example, this guarantees that an adversary cannot issue a signature of knowledge of w on some new message m unless he knows w, even with access to another party who does know w. This is because the signatures issued by other parties do not reveal any information about w, while in order to obtain a valid signature, the adversary must reveal w to our ideal functionality. Although this definition seems to capture the intuition, it does not necessarily give us any hints as to how a signature of knowledge can be constructed. Our second definition helps with that.

Our second definition is a game-style one [22, 3]. This definition requires that a signature of knowledge scheme be in the public parameter model (where the parameters are generated by some trusted process called Setup) and consist of two algorithms, Sign and Verify. Besides the usual correctness property that requires that Verify accept all signatures issued by Sign, we also require that (1) signatures do not reveal anything about the witness; this is captured by requiring that there exist a simulator who can undetectably forge signatures of knowledge without seeing the witness using some trapdoor information about the common parameters; and (2) valid signatures can only be generated by parties who know corresponding witnesses; this is captured by requiring that there exist an extractor who can, using some trapdoor information about the common parameters, extract the witness from any signature of knowledge, even one generated by an adversary with access to the oracle producing simulated signatures. This definition is presented in Section 2.2. (We call this definition *SimExt-security*, for *sim*ulation and *ext*raction.)

We prove that the two definitions are equivalent: namely, a scheme UC-realizes our ideal functionality if and only if it is SimExt-secure.

Our ideal signature of knowledge functionality can be naturally extended to a signature of knowledge of an accepting input to another ideal functionality. For example, suppose that F_{Σ} is the (regular) signature functionality. Suppose w is a signature on the value x under public key PK, issued by the ideal F_{Σ} functionality. Then our functionality F_{SOK} can issue a signature σ on message m, whose meaning is as follows: "The message m is signed by someone who knows w, where w is a signature produced by F_{Σ} under public key PK on message x." In other words, a signature w on message x under public key PK that causes the verification algorithm for F_{Σ} to accept, can be used as a witness for a signature of knowledge. A complication in defining the signature of knowledge functionality this way is that, to be meaningful, the corresponding instance of the F_{Σ} functionality must also be accessible somehow, so that parties can actually obtain signatures under public key PK. Further, for F_{SOK} to be UC-realizable, we must require that the functionality that decides that w is a witness for x, also be UC-realizable. See Section 4 to see how we tackled these definitional issues. As far as we know, this is the first time that an ideal functionality is defined as a function of other ideal functionalities, which may be of independent interest to the study of the UC framework.

Our Constructions. In Section 3, we show how to construct signatures of knowledge for any polynomial-time Turing machine M_L deciding whether w is a valid witness for $x \in L$. We use the fact (proved in Section 2.3) that SimExt-security is a necessary and sufficient notion of security, and give a construction of a SimExt-secure signature of knowledge. Our construction is based on standard assumptions. In the common random string model, it requires a dense cryptosystem [15, 14] and a simulation-sound non-interactive zero-knowledge proof scheme with efficient provers[21, 13] (which can be realized assuming trapdoor permutations for example).

We then show in Section 4 that, given any UC-realizable functionality $\mathcal{F}$ that responds to verification queries and is willing to publish its verification algorithm, the functionality which generates signatures of knowledge of an accepting input to $\mathcal{F}$ is also UC-realizable. We then explain why this yields a delegatable anonymous credential scheme.

The History of the Terminology. The term "signature of knowledge" was introduced by Camenisch and Stadler [7], who use this term to mean a proof of knowledge (more specifically, a Σ-protocol [12]) turned into a signature using the Fiat-Shamir heuristic. Many subsequent papers on group signatures and anonymous credentials used this terminology as well. However, existing literature does not contain definitions of security for the term. Every time a particular construction uses a signature of knowledge as defined by Camenisch and Stadler, the security of the construction is analyzed from scratch, and the term "signature of knowledge" is used more for ease of exposition than as a cryptographic building block whose security properties are well-defined. This frequent informal use of signatures of knowledge indicates their importance in practical constructions and therefore serves as additional motivation of our formal study.

2 Signatures of Knowledge of a Witness for $x \in L$

A signature of knowledge scheme must have two main algorithms, Sign and Verify. The Sign algorithm takes a message and allows anyone holding a witness to a statement $x \in L$ to issue signatures on behalf of that statement. The Verify algorithm takes a message, a statement $x \in L$, and a signature σ, and verifies that the signature was generated by someone holding a witness to the statement.

Signatures of knowledge are essentially a specialized version of noninteractive zero knowledge proofs of knowledge: If a party P can generate a valid signature of knowledge on any message m for a statement $x \in L$, that should mean that, first of all, the statement is true, and secondly, P knows a witness for that statement. This intuitively corresponds to the soundness and extraction properties of a non-interactive proof of knowledge system. On the other hand, just as in a zero-knowledge proof, the signature should reveal nothing about the witness w. We know that general NIZK proof systems are impossible without some common parameters. Thus, our signatures of knowledge will require a setup procedure which outputs shared parameters for our scheme.

Thus, we can define the algorithms in a signature of knowledge schemes as follows: Let $\{Mes_k\}$ be a set of message spaces, and for any language $L \in NP$, let M_L denoted a polynomial time Turing machine which accepts input (x, w) iff w is a witness showing that $x \in L$. Let Setup be an algorithm that outputs public parameters $p \in \{0,1\}^k$ for some parameter k. Let $\mathsf{Sign}(p, M_L, x, w, m)$ be an algorithm that takes as input some public parameters p, a TM M_L for a language L in NP, a value $x \in L$, a valid witness w for x, and $m \in Mes_k$, a message to be signed. Sign outputs a signature of knowledge for instance $x \in L$ on the message m. Let $\mathsf{Verify}(p, M_L, x, m, \sigma)$ be an algorithm that takes as input the values p, M_L, x, the message m, and a purported signature σ, and either accepts or rejects.

2.1 An Ideal Functionality for a Signature of Knowledge

Canetti's Universal Composability framework gives a simple way to specify the desired functionality of a protocol. Furthermore, the UC Theorem guarantees that protocols will work as desired, no matter what larger system they may be operating within. We begin by giving a UC definition of signatures of knowledge.

We begin by recalling Canetti's signature functionality. For a detailed discussion and justification for Canetti's modelling choices see [9].

Note that this functionality is allowed to produce an error message and halt, or quit, if things go wrong. That means that it is trivially realizable by a protocol that always halts. We will therefore only worry about protocols that realize our functionalities *non-trivially*, i.e. never output an error message.

The session id(sid) of $\mathcal{F}_{SIG}$ captures the identity P of the signer; all participants in the protocol with this session id agree that P is the signer. In a signature of knowledge, we do not have one specific signer, so P should not be included in the session id. But all participants in the protocol should agree on the language that they are talking about. Thus, we have a language $L \in \mathbf{NP}$

and a polynomial-time Turing machine M_L and a polynomial p, such that $x \in L$ iff there exists a witness w such that $|w| = p(|x|) \wedge M_L(x, w) = 1$. Let us capture the fact that everyone is talking about the same L by requiring that the session id begin with the description of M_L. As mentioned above, signatures of

$\mathcal{F}_{SIG}$: Canetti's signature functionality

Key Generation Upon receiving a value ($\texttt{KeyGen}$,sid) from some party P, verify that $sid = (P, sid')$ for some sid'. If not, then ignore the request. Else, hand ($\texttt{KeyGen}$,sid) to the adversary. Upon receiving ($\texttt{Algorithms}$, sid, Verify, Sign) from the adversary, where Sign is a description of a PPT ITM, and Verify is a description of a *deterministic* polytime ITM, output ($\texttt{VerificationAlgorithm}$, sid, Verify) to P.

Signature Generation Upon receiving a value ($\texttt{Sign}$,sid, m) from P, let $\sigma \leftarrow \mathsf{Sign}(m)$, and verify that $\mathsf{Verify}(m, \sigma) = 1$. If so, then output ($\texttt{Signature}$, sid, m, σ) to P and record the entry (m, σ). Else, output an error message ($\mathsf{Completeness\ error}$) to P and halt.

Signature Verification Upon receiving a value ($\texttt{Verify}$, $sid, m, \sigma, \mathsf{Verify}'$) from some party V, do: If $\mathsf{Verify}' = \mathsf{Verify}$, the signer is not corrupted, $\mathsf{Verify}(m, \sigma') = 1$, and no entry (m, σ') for any σ' is recorded, then output an error message ($\mathsf{Unforgeability\ error}$) to V and halt. Else, output ($\texttt{Verified}$,$sid, m, \mathsf{Verify}'(m, \sigma)$) to V.

knowledge inherently require some setup. Just as in the key generation interface of $\mathcal{F}_{SIG}$ above, a signature of knowledge functionality ($\mathcal{F}_{SOK}$) setup procedure will determine the algorithm Sign that computes signatures and the algorithm Verify for verifying signatures. However, since anyone who knows a valid witness w can issue a signature of knowledge on behalf of $x \in L$, both Sign and Verify will have to be available to any party who asks for them. In addition, the setup procedure will output algorithms $\mathsf{Simsign}$ and $\mathsf{Extract}$ that we will explain later.

There are three things that the signature generation part of the $\mathcal{F}_{SOK}$ functionality must capture. The first is that in order to issue a signature, the party who calls the functionality must supply (m, x, w) where w is a valid witness to the statement that $x \in L$. This is accomplished by having the functionality check that it is supplied a valid w. The second is that a signature reveals nothing about the witness that is used. This is captured by issuing the formal signature σ via a procedure that does not take w as an input. We will call this procedure $\mathsf{Simsign}$ and require that the adversary provide it in the setup step. Finally, the signature generation step must ensure that the verification algorithm Verify is complete, i.e., that it will accept the resulting signature σ. If it find that Verify is incomplete, $\mathcal{F}_{SOK}$ will output and error message ($\mathsf{Completeness\ error}$) and halt, just as $\mathcal{F}_{SIG}$ does.

The signature verification part of $\mathcal{F}_{SOK}$ should, of course, accept signatures (m, x, σ) if m was previously signed on behalf of $x \in L$, and σ is the resulting signature (or another signature such that $\mathsf{Verify}(m, x, \sigma) = 1$). However, unlike $\mathcal{F}_{SIG}$, just because m was not signed on behalf of x through the signing interface, that does not mean that σ should be rejected, even if the signer is uncorrupted. Recall that anyone who knows a valid witness should be able to generate acceptable signatures! Therefore, the verification algorithm must somehow check that whoever generated σ knew the witness w. Recall that in the setup stage, the adversary provided the algorithm Extract. This algorithm is used to try to extract a witness from a signature σ that was not produced via a call to $\mathcal{F}_{SOK}$. If $\mathsf{Extract}(m, x, \sigma)$ produces a valid witness w, then $\mathcal{F}_{SOK}$ will output the outcome of $\mathsf{Verify}(m, x, \sigma)$. If $\mathsf{Extract}(m, x, \sigma)$ fails to produce a valid witness, and $\mathsf{Verify}(m, x, \sigma)$ rejects, then $\mathcal{F}_{SOK}$ will reject. What happens if $\mathsf{Extract}(m, x, \sigma)$ fails to produce a valid witness, but $\mathsf{Verify}(m, x, \sigma)$ accepts? This corresponds to the case when a signature σ on m on behalf of x was produced without a valid witness w, and yet σ is accepted by Verify. If this is ever the case, then there is an unforgeability error, and so $\mathcal{F}_{SOK}$ should output (Unforgeabilityerror) and halt. Unlike $\mathcal{F}_{SIG}$, here we need not worry about whether the requesting party supplied a correct verification algorithm, since here everyone is on the same page and is always using the same verification algorithm (determined in the setup phase).

$\mathcal{F}_{SOK}(L)$: signature of knowledge of a witness for $x \in L$

Setup Upon receiving a value (`Setup`,sid) from any party P, verify that $sid = (M_L, sid')$ for some sid'. If not, then ignore the request. Else, if this is the first time that (`Setup`,sid) was received, hand (`Setup`,sid) to the adversary; upon receiving (`Algorithms`, sid, Verify, Sign, Simsign, Extract) from the adversary, where Sign, Simsign, Extract are descriptions of PPT TMs, and Verify is a description of a deterministic polytime TM, store these algorithms. Output the stored (`Algorithms`, sid, Sign, Verify) to P.

Signature Generation Upon receiving a value (`Sign`,sid, m, x, w) from P, check that $M_L(x, w) = 1$. If not, ignore the request. Else, compute $\sigma \leftarrow \mathsf{Simsign}(m, x)$, and check that $\mathsf{Verify}(m, x, \sigma) = 1$. If so, then output (`Signature`,sid, m, x, σ) to P and record the entry (m, x, σ). Else, output an error message (Completeness error) to P and halt.

Signature Verification Upon receiving a value (`Verify`, sid, m, x, σ) from some party V, do: If (m, x, σ') is stored for some σ', then output (`Verified`,$sid, m, x, \sigma, \mathsf{Verify}(m, x, \sigma)$) to V. Else let $w \leftarrow \mathsf{Extract}(m, x, \sigma)$; if $M_L(x, w) = 1$, output (`Verified`,$sid, m, x, \sigma, \mathsf{Verify}(m, x, \sigma)$) to V. Else if $\mathsf{Verify}(m, x, \sigma) = 0$, output (`Verified`,$sid, m, x, \sigma, 0$) to V. Else output an error message (Unforgeability error) to V and halt.

In the UC framework, each instance of the ideal functionality is associated with a unique sid, and it ignores all queries which are not addressed to this sid. Since our $\mathcal{F}_{SOK}$ functionalities require that $sid = M_L \circ sid'$, this means that each $\mathcal{F}_{SOK}$ functionality handles queries for exactly one language.

Now consider the following language U.

Definition 1 (Universal language). *For polynomial p, define universal language U_p s.t. x would contain a description of a Turing machine M and an instance x' such that $x \in U_p$ iff there exists w s.t. $M(x', w)$ halts and accepts in time at most $p(|x|)$.*

Notice that $\mathcal{F}_{SOK}(U_p)$ allows parties to sign messages on behalf of any instance x of any language L which can be decided in non-deterministic $p(|x|)$ time. Thus, if we have Setup, Sign, and Verify algorithms which realize $\mathcal{F}_{SOK}(U_p)$, we can use the same algorithms to generate signatures of knowledge for all such instances and languages. In particular, this means we do not need a separate setup algorithm (in implementation, a separate CRS or set of shared parameters) for each language. Readers familiar with UC composability may notice that any protocol which realizes $\mathcal{F}_{SOK}(U_p)$ for all polynomials p will also realize the multisession extension of $\mathcal{F}_{SOK}$. For more information, see full version.

2.2 A Definition in Terms of Games

We now give a second, games style definition for signatures of knowledge. We will show that this definition is equivalent to (necessary and sufficient for) the UC definition given in the previous section. Informally, a signature of knowledge is SimExt-secure if it is correct, simulatable and extractable.

The **correctness** property is similar to that of a traditional signature scheme. It requires that any signature issued by the algorithm Sign should be accepted by Verify.

The **simulatability** property requires that there exist a simulator which, given some trapdoor information on the parameters, can create valid signatures without knowing any witnesses. This captures the idea that signatures should reveal nothing about the witness used to create them. Since the trapdoor must come from somewhere, the simulator is divided into Simsetup that generates the public parameters (possibly from some different but indistinguishable distribution) together with the trapdoor, and Simsign which then signs using these public parameters. We require that no adversary can tell that he is interacting with Simsetup and Simsign rather than Setup and Sign.

The **extraction** property requires that there exist an extractor, which given a signature of knowledge for an $x \in L$, and appropriate trapdoor information, can produce a valid witness showing $x \in L$. This captures the idea that it should be impossible to create a valid signature of knowledge without knowing a witness. In defining the extraction property, we require that any adversary that interacts with the simulator Simsetup and Simsign (rather than the Setup and Sign) not be able to produce a signature from which the extractor cannot extract a witness. The reason that in the definition, the adversary interacts with Simsetup instead of Setup is because the extractor needs a trapdoor to be able to extract. Note that it also interacts with Simsign instead of Sign. The adversary could run Sign itself, so access to Simsign gives it a little bit of extra power.

Definition 2 (SimExt-security). *Let L be the language defined by a polynomial-time Turing machine M_L as explained above, such that all witnesses for $x \in L$ are of known polynomial length $p(|x|)$. Then* (Setup, Sign, Verify) *constitute a SimExt-secure signature of knowledge of a witness for L, for message space $\{Mes_k\}$ if the following properties hold:*

Correctness *There exists a negligible function ν such that for all $x \in L$, valid witnesses w for x(i.e. witnesses w such that $M_L(x, w) = 1$), and $m \in Mes_k$*

$$\Pr[p \leftarrow \mathsf{Setup}(1^k); \sigma \leftarrow \mathsf{Sign}(p, M_L, x, w, m) \; : \; \mathsf{Verify}(p, M_L, x, m, \sigma) = \mathit{Accept}] = 1 - \nu(k)$$

Simulatability *There exists a polynomial time simulator consisting of algorithms* Simsetup *and* Simsign *such that for all probabilistic polynomial-time adversaries $\mathcal{A}$ there exists a negligible functions ν such that for all polynomials f, for all k, for all auxiliary input $s \in \{0,1\}^{f(k)}$*

$$\left| \begin{array}{l} \Pr[(p,\tau) \leftarrow \mathsf{Simsetup}(1^k); b \leftarrow \mathcal{A}^{\mathsf{Sim}(p,\tau,\cdot,\cdot,\cdot,\cdot)}(s,p) \; : \; b = 1] \\ \; - \; \Pr[p \leftarrow \mathsf{Setup}(1^k); b \leftarrow \mathcal{A}^{\mathsf{Sign}(p,\cdot,\cdot,\cdot,\cdot)}(s,p) \; : \; b = 1] \end{array} \right| = \nu(k)$$

where the oracle Sim *receives the values (M_L, x, w, m) as inputs, checks that the witness w given to it was correct and returns $\sigma \leftarrow \mathsf{Simsign}(p, \tau, M_L, x, m)$. τ is the additional trapdoor value that the simulator needs in order to simulate the signatures without knowing a witness.*

Extraction *In addition to* (Simsetup, Simsign)*, there exists an extractor algorithm* Extract *such that for all probabilistic polynomial time adversaries $\mathcal{A}$ there exists a negligible function ν such that for all polynomials f, for all k, for all auxiliary input $s \in \{0,1\}^{f(k)}$*

$$\Pr\left[(p,\tau) \leftarrow \mathsf{Simsetup}(1^k); (M_L, x, m, \sigma) \leftarrow \mathcal{A}^{\mathsf{Sim}(p,\tau,\cdot,\cdot,\cdot,\cdot)}(s,p); \; w \leftarrow \mathsf{Extract}(p, \tau, M_L, x, m, \sigma) : \; M_L(x,w) \vee (M_L, x, m, w) \in Q \vee \neg\mathsf{Verify}(p, M_L, x, m, \sigma)\right] = 1 - \nu(k)$$

where Q denotes the query tape which lists all previous queries (M_L, x, m, w) $\mathcal{A}$ has sent to the oracle Sim*.*

Note that the above definition captures, for example, the following intuition: suppose that Alice is the only one in the world who knows the witness w for $x \in L$, and it is infeasible to compute w. Then Alice can use x as her signing public key, and her signature σ on a message m can be formed using a signature of knowledge w. We want to make sure that the resulting signature should be existentially unforgeable against chosen message attacks [17]. Suppose it is not. Then there is a forger who can output (m, σ), such that σ is accepted by the verification algorithm without a query m to Alice. Very informally, consider the following four games:

Adversary vs. Alice: The parameters are generated by Setup. Alice chooses a random x, w pair and publishes x. The adversary sends Alice messages to be signed and Alice responds to each using x, w and Sign. Adversary outputs a purported forgery. Let p_0 be the probability that the forgery is successful.

Adversary vs. Simulator: The simulator generates parameters using Simsetup. The simulator chooses a random x, w pair and publishes x. The adversary sends the simulator messages to be signed, and he responds using x, w and Sim. The adversary outputs a purported forgery. Let p_1 be the probability that the forgery is successful.

Adversary vs. Extractor: The extractor generates parameters using Simsetup. He then chooses a random x, w pair and publishes x. The adversary sends the simulator messages to be signed, and he responds using x, w and Sim. The adversary outputs a purported forgery. The extractor runs Extract to obtain a potential witness w. Let p_2 be the probability that w is a valid witness.

Adversary vs. Reduction: The reduction is given and instance x, which it publishes. It then generates parameters using Simsetup. The adversary sends messages to be signed, and the reduction responds using x and $\mathsf{Simsign}$. The adversary outputs a purported forgery. The reduction runs $\mathsf{Extract}$ to obtain w. Let p_3 be the probability that w is a valid witness.

By the simulatability property, the difference between p_0 and p_1 must be negligible. By the extraction property, the difference between p_1 and p_2 must be negligible. Since Sim ignores w and runs $\mathsf{Simsign}$, p_2 and p_3 must be identical. Thus, generating forgeries is at least as hard as deriving a witness w for a random instance x. If the algorithm used to sample (x, w) samples hard instances with their witnesses, then we know that the probability of forgery is negligible. For a formal proof see full version.

2.3 Equivalence of the Definitions

As was mentioned above, signatures of knowledge cannot exist without some trusted setup procedure which generates shared parameters. In the UC model, shared parameters are captured by the $\mathcal{F}_{CRS}^{D}$ functionality [8]. This functionality generates values from a given distribution D (the desired distribution of shared parameters), and makes them available for all parties in the protocol. Thus, protocols requiring shared parameters can be defined in the $\mathcal{F}_{CRS}$-hybrid model, where real protocols are given access to the ideal shared parameter functionality.

Formally, the $\mathcal{F}_{CRS}^{D}$ functionality receives queries of the form $(\texttt{CRS}, sid)$ from a party P. If a value v for this sid has not been stored, it chooses a random value v from distribution D and stores it. It returns $(\texttt{CRS}, sid, v)$ to P and also sends $(\texttt{CRS}, sid, v)$ to the adversary.

Let $\Sigma = (\mathsf{Setup}, \mathsf{Sign}, \mathsf{Verify})$ be a signature of knowledge scheme. Let k be the security parameter. We define a F_{CRS}^{D}-hybrid signature of knowledge protocol π_Σ, where D is the distribution of $\mathsf{Setup}(1^k)$.

When a party P running π_Σ receives an input $(\texttt{Setup}, sid)$ from the environment, it checks that $sid = (M_L, sid')$ for some sid'. If not it ignores the request. It then queries the $\mathcal{F}_{CRS}$ functionality, receives $(\texttt{CRS}, v)$, and stores v. It returns $(\texttt{Algorithms}, sid, \mathsf{Sign}(v, M_L, \cdot, \cdot, \cdot), \mathsf{Verify}(v, M_L, \cdot, \cdot))$ to the environment.

When P receives a request $(\texttt{Sign}, sid, m, x, w)$ from the environment, it retrieves the stored v. It checks that $M_L(x, w) = 1$. If not, it ignores the request, otherwise it returns $(\texttt{Signature}, sid, m, x, \mathsf{Sign}(v, M_L, x, w, m))$. When P receives a request $(\texttt{Verify}, sid, m, x, \sigma)$ from the environment, it again retrieves the stored v, and then returns $(\texttt{Verified}, sid, m, x, \sigma, \mathsf{Verify}(v, M_L, x, m, \sigma))$.

Theorem 1. *For any polynomial p, π_Σ UC-realizes $\mathcal{F}_{SOK}(U_p)$ in the $\mathcal{F}_{CRS}^{D}$-hybrid model if and only if Σ is SimExt-secure.*

Proof. (Sketch: See the full version for the full proof.) Suppose that Σ is SimExt-secure. Then let us show that π_Σ UC-realizes $\mathcal{F}_{SOK}(U_p)$. Consider the ideal adversary (simulator) S that works as follows: Upon receiving $(\texttt{Setup}, sid)$ from $\mathcal{F}_{SOK}$, S will parse $sid = (M_U, sid')$. It obtains $(p, \tau) \leftarrow \mathsf{Simsetup}(1^k)$ and sets $\mathsf{Sign} = \mathsf{Sign}(p, \cdot, \cdot, \cdot, \cdot)$ (so Sign will have four inputs: the language M_L — note that since we are realizing $\mathcal{F}_{SOK}(U_p)$, any instance will start with M_L,— the instance $x \in L$, the witness w, and the message m), $\mathsf{Verify} = \mathsf{Verify}(p, \cdot, \cdot, \cdot, \cdot)$,

$\mathsf{Simsign} = \mathsf{Simsign}(p, \tau, \cdot, \cdot, \cdot, \cdot)$, and $\mathsf{Extract} = \mathsf{Extract}(p, \tau, \cdot, \cdot, \cdot, \cdot)$. Finally, it sends ($\mathtt{Algorithms}$,$sid$, Sign, Verify, $\mathsf{Simsign}$, $\mathsf{Extract}$) back to $\mathcal{F}_{SOK}$. When the adversary A queries the $\mathcal{F}^{D}_{CRS}$ functionality, S outputs p.

Let Z be any environment and A be an adversary. We wish to show that Z cannot distinguish interactions with A and π_Σ from interactions with S and $\mathcal{F}_{SOK}$. Let us do that in two steps. First, we show that the event E that $\mathcal{F}_{SOK}$ halts with an error message has negligible probability. Next, we will show that, conditioned on E not happening, Z's view in its interaction with S and $\mathcal{F}_{SOK}$ is indistinguishable from its view in interactions with A and π_Σ.

There are two types of errors that lead to event E: $\mathcal{F}_{SOK}$ halts with $\mathsf{Completeness\ error}$ or $\mathsf{Unforgeability\ error}$. The only way to induce a completeness error is to cause Verify to reject a signature issued by $\mathsf{Simsign}$, which contradicts either the simulatability or the correctness requirement. The only way to induce an unforgeability error is to cause Verify to accept a signature which was not issued by $\mathsf{Simsign}$ and from which no witness can be extracted. This contradicts the extractability requirement.

We have shown that the probability of event E is negligible. Conditioned on $\bar{E}$, Z's view when interacting with $\mathcal{F}_{SOK}$ and S is indistinguishable from its view when interacting with a real adversary A and the real protocol π_Σ, because if it were distinguishable, then this would contradict the simulatability requirement.

The converse is fairly straightforward and appears in the full version. $\square$

3 Construction

Here we present Σ, a construction of a SimExt-secure signature of knowledge. By Theorem 1, this also implies a protocol π_Σ that UC-realizes the $\mathcal{F}_{SOK}$ functionality presented in Section 2.1.

Our construction has two main building blocks: CPA secure dense cryptosystems [15, 14] and simulation-sound non-interactive zero knowledge proofs [21, 13]. Let $(\mathcal{G}, \mathsf{Enc}, \mathsf{Dec})$ be a dense cryptosystem, and let $(\mathsf{NIZKProve}, \mathsf{NIZKSimsetup}, \mathsf{NIZKSim}, \mathsf{NIZKVerify})$ be a simulation-sound non-interactive zero-knowledge proof system.

Setup Let p be a common random string. Parse p as follows: $p = PK \circ \rho$, where PK is a k-bit public key of our cryptosystem.

Signature Generation In order to sign a message $m \in Mes_k$ using knowledge of witness w for $x \in L$, let $c = \mathsf{Enc}(PK, (m, w), R)$, where R is the randomness needed for the encryption process; let $\pi \leftarrow \mathsf{NIZKProve}(\rho, (m, M_L, x, c, PK), (\exists (w, R) : c = \mathsf{Enc}(PK, (m, w), R) \wedge M_L(x, w)), (w, R))$. Output $\sigma = (c, \pi)$.

Verification In order to verify a signature of knowledge of witness w for $x \in L$, $\sigma = (c, \pi)$, run $\mathsf{NIZKVerify}(\rho, \pi, (m, M_L, x, c, PK), (\exists (w, R) : c = \mathsf{Enc}(PK, (m, w), R) \wedge M_L(x, w)))$.

Intuitively, the semantic security of the cryptosystem together with the zero knowledge property of the proof system ensure that the signature reveals no information about the witness. The simulation soundness property of the proof system means that the adversary cannot prove false statements. Thus any signature that verifies must include a ciphertext which is an encryption of the given message and of a valid witness. Clearly, if he is interacting only with a simulator who does not know any witnesses, this implies that the adversary should know the witness. Further, by simulatability, the adversary cannot gain any advantage by communicating with valid signers. The key is that the witness and message are encrypted together, and there is a single proof that the encryption is correct. Thus it is not possible to simply replace the message with a different one.

Theorem 2. *The construction above is a SimExt-secure signature of knowledge.*

Proof. (Sketch: See full version for full proof) First we argue simulatability. In the $\mathsf{Simsetup}$ phase, our simulator will choose a key pair (PK, SK) of the dense cryptosystem, and will obtain the string ρ together with trapdoor τ' by running $\mathsf{NIZKSimsetup}$. In the $\mathsf{Simsign}$ phase, the simulator will always let c be the encryption of $0^{|m|+l_L}$, and will create (fake) proof π by invoking $\mathsf{NIZKSim}$.

We show that the resulting simulation is successful using a two-tier hybrid argument. First, by the unbounded zero-knowledge property of the underlying NIZK proof system, signatures obtained by replacing calls to $\mathsf{NIZKProve}$ by calls to $\mathsf{NIZKSim}$ will be distributed indistinguishably from real signatures. Second, by semantic security of the dense cryptosystem used, using $c \leftarrow \mathsf{Enc}(PK, (m, w))$ versus $c \leftarrow \mathsf{Enc}(PK, (0^{|m|+l_L}))$ results in indistinguishable distributions.

Second, let us argue extraction. Recall that, as part of the trapdoor τ, $\mathsf{Simsetup}$ above retains SK, the secret key for the cryptosystem. The extractor simply decrypts the c part of the signature σ to obtain the witness w. By the simulation-soundness property of the underlying NIZK proof system, no adversary can produce a signature acceptable to the Verify algorithm without providing c that decrypts to a correct witness w. [1] □

4 $\mathcal{F}_{SOK}$ for Generalized Languages, and Applications

Recall from the introduction that a signature of knowledge may be used in order to construct a group signature scheme. Let PK_s be the public signing key of the group manager, and suppose that the group manager can sign under this public key (using the corresponding secret key SK_s). Let PK_E be a public encryption key such that the anonymity revocation manager knows the corresponding secret key SK_E. A user must pick a secret key x and a public key $p = f(x)$ where f is some one-way function. She then obtains a group membership certificate $g = \sigma_{PK_s}(p)$, the group manager's signature on her public key. In order to sign on behalf of the group, the user encrypts her public key and obtains a ciphertext

[1] Note that a CPA secure cryptosystem is sufficient: We only need the security of the encryption scheme to guarantee that the encryptions of $0^{|m|+l_L}$ generated by the simulator are indistinguishable from encryptions of valid witness/message pairs. This is only necessary in the proof of simulatability, in a scenario where pairs are encrypted but never decrypted.

$c = Enc(PK_E, p; R)$, where R is the randomness used for encryption. Finally, her group signature on message m is a signature of knowledge of (x, p, g, R) such that $c = Enc(PK_E, p, R)$, $p = f(x)$, and g is a valid signature on p under PK_s.

Now let us consider more closely the language L used in the signature of knowledge. In the example above, $c \in L$ and (x, p, g, R) is the witness. This language is determined by the parameters of the system, (f, PK_s, PK_E). This is not a general language, but instead it depends on the system parameters, which in turn depend on three other building blocks, a one-way function, an encryption scheme and a signature scheme. We want to show that even in this context, the use of a signature of knowledge has well-understood consequences for the security of the rest of the system.

To that end, we consider signatures of knowledge for languages that are defined by secure functionalities realizing particular tasks. In this example, this corresponds to the one-way function, encryption and signing functionalities. Encryption is used to incorporate the encrypted identity, c, of the signer into her group signature. A signing functionality is used to issue group membership certificates, g, to individual group members. Finally, we have a one-way function f that takes a user's secret x and maps it to her public p.

We could choose a specific realization of each of these primitives, combine these realizations, and use the resulting TM to define our language L for a signature of knowledge as described in Section 2. However, we would like to be able to define an abstract signature of knowledge functionality whose language is defined by ideal functionalities and not dependent on any specific realizations.

In this section, we wish to create a framework where, given ideal functionalities $\mathcal{F}_f$, $\mathcal{F}_{Enc}$ and $\mathcal{F}_{\Sigma}$ for these three primitives, we can define a signature of knowledge functionality $\mathcal{F}_{SOK}$ for the language L, where L is defined in terms of the outputs of functionalities $\mathcal{F}_f$, $\mathcal{F}_{Enc}$, and F_{Σ}. Such $\mathcal{F}_{SOK}$ can be used to realize group signatures as above, as well as other cryptographic protocols.

First, in Section 4.1, we will characterize functionalities that define such generalized languages L. These are functionalities which, when they receive an input (x, w), verify that this is indeed an accepting input, i.e. that w constitutes a witness for $x \in L$. In Section 4.2, we will define $\mathcal{F}_{SOK}(\mathcal{F}_0)$, a signature of knowledge of an accepting input to one ideal functionality, $\mathcal{F}_0$. Then, we prove Theorem 3: given a SimExt-secure scheme, $\mathcal{F}_{SOK}(\mathcal{F}_0)$ is UC-realizable in the CRS model if and only if $\mathcal{F}_0$ is UC-realizable. In the full version we explain several generalizations of this ideal functionality which allow us to apply Theorem 3 to group signatures, delegatable credentials, and other similar scenarios.

As far as we know, prior literature on the UC framework did not address the issues of defining an ideal functionality as an extension of another ideal functionality or of a set of other functionalities. (In contrast, it addressed the case when a *real* protocol used an ideal functionality as a sub-routine.)

4.1 Explicit Verification Functionalities

Only certain functionalities make sense as a language for a signature of knowledge. In particular, they need to allow us to determine whether a given element is in the language given a potential witness: we call this "verification." We also assume that everyone knows how language membership is determined. Thus, we also require that the functionality be willing to output code which realizes its verification procedure. In this section, we formally define the functionalities which can be used to define the language for a signature of knowledge.

Consider Canetti's signature functionality $\mathcal{F}_{SIG}$. Once the key generation algorithm has been run, this functionality defines a language: namely, the language of messages that have been signed. A witness for membership in such a language is the signature σ. In a `Verify_` query this functionality will receive (m, σ) and will accept if m has been signed and $\mathsf{Verify}(m, \sigma) = \mathit{Accept}$, where Verify is the verification algorithm supplied to $\mathcal{F}_{SIG}$ by the ideal adversary S. Moreover, if it so happens that $\mathsf{Verify}(m, \sigma)$ accepts while m has not been signed, or if it is the case that $\mathsf{Verify}(m, \sigma)$ rejects a signature generated by $\mathcal{F}_{SIG}$, $\mathcal{F}_{SIG}$ will halt with an error. $\mathcal{F}_{SIG}$ is an example of a verification functionality, defined below:

Definition 3 ((Explicit) Verification functionality). *A functionality $\mathcal{F}$ is a verification functionality if (1) there exists some $start(\mathcal{F})$ query such that $\mathcal{F}$ ignores all queries until it receives a start query; (2) during the start query $\mathcal{F}$ obtains from the ideal adversary $\mathcal{S}$ a deterministic polynomial-time verification algorithm* Verify*; (3) in response to (*`Verify`*,sid, $input$, $witness$) queries, $\mathcal{F}$ either responds with the output of* $\mathsf{Verify}(input, witness)$ *or halts with an error. $\mathcal{F}$ is an explicit verification functionality if, once a $start(\mathcal{F})(sid)$ query has taken place, it responds to a query (*`VerificationAlgorithm`*,sid) from any party P by returning the algorithm* Verify*.*

Note that $start(\mathcal{F})$ is a specific command that depends on the functionality $\mathcal{F}$. For example, if $\mathcal{F}$ is a signature functionality, $start(\mathcal{F})$ =Keygen. If $\mathcal{F}$ is another signature of knowledge functionality, $start(\mathcal{F})$ =Setup.

An explicit verification functionality not only defines a language L, but also makes available a Turing machine M_L deciding whether w is a witness for $x \in L$.

4.2 Signatures of Knowledge of an Accepting Input to $\mathcal{F}_0$

Let $\mathcal{F}_0$ be any explicit verification functionality. (Our running example is Canetti's signature functionality, or our own $\mathcal{F}_{SOK}$ functionality, augmented so that it responds to `VerificationAlgorithm` queries with the Verify algorithm obtained from the ideal adversary.) We want to build a signature of knowledge functionality $\mathcal{F}_{SOK}(\mathcal{F}_0)$ that incorporates $\mathcal{F}_0$. It creates an instance of $\mathcal{F}_0$ and responds to all the queries directed to that instance. So, if $\mathcal{F}_0$ is a signature functionality, then $\mathcal{F}_{SOK}(\mathcal{F}_0)$ will allow some party P to run key generation and signing, and will also allow anyone to verify signatures. In addition, any party in possession of (x, w) such that $\mathcal{F}_0$'s verification interface will accept (x, w), can sign on behalf of the statement "There exists a value w such that $\mathcal{F}_0(sid_0)$ accepts (x, w)." For example, if $\mathcal{F}_0$ is a signing functionality, m is a message, and σ_0 is a signature on m created by P with session id sid_0, then through $\mathcal{F}_{SOK}(\mathcal{F}_0)$, any party knowing (m, σ_0) can issue a signature σ_1, which is a signature of knowledge of a signature σ_0 on m, where σ_0 was created by signer P. Moreover, any party can verify the validity of σ_1.

To define $\mathcal{F}_{SOK}(\mathcal{F}_0)$, we start with our definition of $\mathcal{F}_{SOK}(L)$ and modify it in a few places. In the protocol description below, these places are underlined.

The main difference in the setup, signature generation, and signature verification interfaces is that the TM M_L that decides whether w is a valid witness for $x \in L$, is no longer passed to the functionality $\mathcal{F}_{SOK}$. Instead, language membership is determined by queries to the verification procedure of $\mathcal{F}_0$, as well as by an algorithm M_L that $\mathcal{F}_0$ returns when asked to provide its verification algorithm. $(\mathsf{Sign}, \mathsf{Verify}, \mathsf{Simsign}, \mathsf{Extract})$ returned by the adversary now take M_L as input. M_L is supposed to be an algorithm that UC-realizes the verification procedure of $\mathcal{F}_0$. Note, however, that just because $M_L(x, w)$ accepts, does not mean that $\mathcal{F}_0$'s verification procedure necessarily accepts. Instead $\mathcal{F}_{SOK}$ expects that $M_L(x, w)$ accepts iff $\mathcal{F}_0$ accepts, and should $\mathcal{F}_{SOK}$ be given (x, w) where this is not the case, $\mathcal{F}_{SOK}$ will output an error message ($\mathsf{Error\ with}\ \mathcal{F}_0$) and halt.

The setup procedure of $\mathcal{F}_{SOK}(\mathcal{F}_0)$ differs from that of $\mathcal{F}_{SOK}(L)$ in two places. First, it used to check that the session id contains the description M_L of the language L; instead now it checks that it contains a description of the functionality $\mathcal{F}_0$ and a session id sid_0 with which $\mathcal{F}_0$ should be invoked. Second, it must now invoke $\mathcal{F}_0$ to determine the language L and the Turing machine M_L (see below).

The signing and verification procedures of $\mathcal{F}_{SOK}(\mathcal{F}_0)$ differs from that of $\mathcal{F}_{SOK}(L)$ only in that, instead of just checking that $M_L(x, w) = 1$, they check that $\mathcal{F}_0$ accepts (x, w) and that M_L faithfully reflects what $\mathcal{F}_0$ does.

Let us explain how the language L is determined. During the first setup query, $\mathcal{F}_{SOK}$ must somehow determine the set of accepted (x, w), i.e., get the language L. To that end, it creates an instance of $\mathcal{F}_0$, and runs the start query for $\mathcal{F}_0$. It also queries $\mathcal{F}_0$ to obtain its verification algorithm M_L. We describe how this is done separately by giving a procedure we call GetLanguage($\mathcal{F}_0, sid_0$), as a subroutine of the setup phase of $\mathcal{F}_{SOK}$.

Note that this instance of $\mathcal{F}_0$ is created *inside* of $\mathcal{F}_{SOK}$, and outside parties cannot access it directly. Instead, if they want to use $\mathcal{F}_0$ and send a query to it of the form $(query, sid_0, data)$, they should instead query $\mathcal{F}_{SOK}$ with a query of the form $(\mathcal{F}_0\text{-}query, sid, data)$, where $sid = (sid_0, sid_1)$ is the session id of $\mathcal{F}_{SOK}$. We specify this more rigorously in the actual description of $\mathcal{F}_{SOK}(\mathcal{F}_0)$. Note that $\mathcal{F}_{SOK}$ will ignore any queries until the first setup query — this is done so that one cannot query $\mathcal{F}_0$ before it is actually created.

Also note that $\mathcal{F}_0$ may require input from the adversary. Whenever this is the case, the messages that $\mathcal{F}_0$ wants to send to the adversary are forwarded to the adversary, and the adversary's responses are forwarded back to $\mathcal{F}_0$.

Finally, we want $\mathcal{F}_{SOK}(\mathcal{F}_0)$ itself to be a explicit verification functionality (as explained in Section 4.1), and so it must be able to respond to queries asking it to provide its verification algorithm.

Theorem 3. *Let $\mathcal{F}_0$ be an explicit verification functionality. Assuming SimExt-secure signatures of knowledge, $\mathcal{F}_{SOK}(\mathcal{F}_0)$ is nontrivially UC-realizable in the $\mathcal{F}^D_{CRS}$ hybrid model iff $\mathcal{F}_0$ is nontrivially UC-realizable in the $\mathcal{F}^D_{CRS}$ hybrid model,* *where we consider a realization to be nontrivial if it never halts with an error message.* For the proof see the full version.

$\mathcal{F}_{SOK}(\mathcal{F}_0)$**: signature of knowledge of an accepting input to** $\mathcal{F}_0$

For any *sid*, ignore any message received prior to (`Setup`, *sid*).

Setup Upon receiving a value (`Setup`,*sid*) from any party P, verify that $sid = (\mathcal{F}_0, sid_0, sid_1)$ for some sid_0, sid_1. If not, then ignore the request. Else, if this is the first time that (`Setup`,*sid*) was received, let $M_L = \mathsf{GetLanguage}(\mathcal{F}_0, sid_0)$, store M_L, and hand (`Setup`,*sid*) to the adversary; upon receiving (`Algorithms`, *sid*, Verify, Sign, Simsign, Extract) from the adversary, where Sign, Simsign, Extract are descriptions of PPT ITMs, and Verify is a description of a deterministic polytime ITM, store these algorithms. Output the (`Algorithms`, *sid*, $\mathsf{Sign}(M_L, \cdot, \cdot, \cdot)$,$\mathsf{Verify}(M_L, \cdot, \cdot, \cdot)$) to P.

Signature Generation Upon receiving a value (`Sign`,sid, m, x, w) from P, check that $\mathcal{F}_0$ accepts (`Verify`,sid_0, x, w) when invoked by P. If not, ignore the request. Else, if $M_L(x, w) = 0$, output an error message (Error with $\mathcal{F}_0$) to P and halt. Else, compute $\sigma \leftarrow \mathsf{Simsign}(M_L, m, x)$, and verify that $\mathsf{Verify}(M_L, m, x, \sigma) = 1$. If so, then output (`Signature`,sid, m, x, σ) to P and record the entry (m, x, σ). Else, output an error message (Completeness error) to P and halt.

Signature Verification Upon receiving a value (`Verify`,sid, m, x, σ) from some party V, do: If (m, x, σ') is stored for some σ', then output (`Verified`,$sid, m, x, \sigma, \mathsf{Verify}(m, x, \sigma)$) to V. Else let $w \leftarrow \mathsf{Extract}(M_L, m, x, \sigma)$. If $M_L(x, w) = 1$: if $\mathcal{F}_0$ does not accept (`Verify`,sid_0, x, w), output and error message (Error with $\mathcal{F}_0$) to P and halt; else output (`Verified`,$sid, m, x, \sigma, \mathsf{Verify}(M_L, m, x, \sigma)$) to V. Else if $\mathsf{Verify}(M_L, m, x, \sigma) = 0$, output (`Verified`,$sid, m, x, \sigma, 0$) to V. Else output an error message (Unforgeability error) to V and halt.

Additional routines:

GetLanguage($\mathcal{F}_0, sid_0$**)** Create an instance of $\mathcal{F}_0$ with session id sid_0. Send to $\mathcal{F}_0$ the message ($start(\mathcal{F}_0), sid_0$) on behalf of P, the calling party. Send to $\mathcal{F}_0$ the message (`VerificationAlgorithm`,sid_0). In response, receive from $\mathcal{F}_0$ the message (`VerificationAlgorithm`,sid_0, M). Output M.

Queries to $\mathcal{F}_0$ Upon receiving a message ($\mathcal{F}_0$-*query*, sid_0, sid_1, *data*) from a party P, send (*query*, sid_0, *data*) to $\mathcal{F}_0$ on behalf of P. Upon receiving (*response*, sid_0, *data*) from $\mathcal{F}_0$, forward ($\mathcal{F}_0$-*response*, *sid*, *data*) to P.

$\mathcal{F}_0$**'s interactions with the adversary** When $\mathcal{F}_0$ wants to send (*command*, sid_0, *data*) to the adversary, give to the adversary the message ($\mathcal{F}_0$-*command*, *sid*, sid_0, *data*). When receive a message ($\mathcal{F}_0$-*header*, *sid*, sid_0, *data*) from the adversary, give (*header*, sid_0, *data*) to $\mathcal{F}_0$ on behalf of the adversary.

Providing the verification algorithm Upon receiving a message (`VerificationAlgorithm`,*sid*) from any party P, output (`VerificationAlgorithm`,$sid, \mathsf{Verify}(M_L, \cdot, \cdot, \cdot)$ to P.

Acknowledgments: The authors wish to thank Jan Camenisch, Mira Meyerovich, and Leo Reyzin for helpful comments and discussion. Melissa Chase is supported by NSF grant CNS-0374661 and NSF Graduate Research Fellowship. Anna Lysyanskaya is supported by NSF CAREER grant CNS-0374661.

References

1. G. Ateniese, J. Camenisch, M. Joye, and G. Tsudik. A practical and provably secure coalition-resistant group signature scheme. In *CRYPTO 2000*, 255–270.
2. M. Bellare, D. Micciancio, and B. Warinschi. Foundations of group signatures: Formal definitions, simplified requirements, and a construction based on general assumptions. In *EUROCRYPT 2003*, pages 614–629.
3. M. Bellare and P. Rogaway. The game-playing technique. `http://eprint.iacr.org/2004/331`, 2004.
4. J. Benaloh and M. de Mare. One-way accumulators: A decentralized alternative to digital signatures. In *EUROCRYPT '93*, pages 274–285.
5. D. Boneh, X. Boyen, and H. Shacham. Short group signatures. In *CRYPTO 2004*, pages 41–55.
6. J. Camenisch and A. Lysyanskaya. Efficient non-transferable anonymous multi-show credential system with optional anonymity revocation. In *EUROCRYPT 2001*, pages 93–118.
7. J. Camenisch and M. Stadler. Efficient group signature schemes for large groups. In *CRYPTO '97*, pages 410–424.
8. R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.
9. R. Canetti. Universally composable security: A new paradigm for cryptographic protocol. `http://eprint.iacr.org/2000/067`, 2005.
10. D. Chaum. Security without identification: Transaction systems to make big brother obsolete. *Communications of the ACM*, 28(10):1030–1044, Oct. 1985.
11. D. Chaum and E. van Heyst. Group signatures. In *EUROCRYPT '91*, 257–265.
12. R. Cramer. *Modular Design of Secure yet Practical Cryptographic Protocol*. PhD thesis, University of Amsterdam, 1997.
13. A. de Santis, G. di Crescenzo, R. Ostrovsky, G. Persiano, and A. Sahai. Robust non-interactive zero knowledge. In *CRYPTO 2001*, pages 566–598.
14. A. de Santis, G. di Crescenzo, and G. Persiano. Necessary and sufficient assumptions for non-interactive zero-knowledge proofs of knowledge for all NP relations. In *ICALP 2000*, pages 451–462.
15. A. de Santis and G. Persiano. Zero-knowledge proofs of knowledge without interaction (extended abstract). In *FOCS 1992*, pages 427–436.
16. Y. Dodis, A. Kiayias, A. Nicolosi, and V. Shoup. Anonymous identification in ad hoc groups. In *EUROCRYPT 2004*, pages 609–626.
17. S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308.
18. A. Lysyanskaya. *Signature schemes and applications to cryptographic protocol design*. PhD thesis, Massachusetts Institute of Technology, Sept. 2002.
19. A. Lysyanskaya, R. Rivest, A. Sahai, and S. Wolf. Pseudonym systems. In *Selected Areas in Cryptography*.
20. R. L. Rivest, A. Shamir, and Y. Tauman. How to leak a secret. In *ASIACRYPT 2001*, pages 552–565.
21. A. Sahai. Non-malleable non-interactive zero knowledge and adaptive chosen-ciphertext security. In *FOCS 1999*, pages 543–553. IEEE, 1999.
22. V. Shoup. Sequences of games: a tool for taming complexity in security proofs. `http://eprint.iacr.org/2004/332`, 2004.