On Signatures of Knowledge

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

August 8, 2006

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.

Keywords: signature schemes, NIZK, proof of knowledge, UC, anonymous credentials

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 $\mathsf{Verify}(\phi, m, \sigma) = Accept$, 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 very 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 $M_L(x, w) = Accept$. 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 [RST01]. 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. [DKNS04] using cryptographic accumulators [BdM94], 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 [CvH91, CS97, ACJT00, BMW03, BBS04]. 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: $c = Enc(PK_E, f(x); R)$ 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 [ACJT00, BBS04].

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. [1] 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 [Cha85]. More recently this problem was raised in the anonymous credentials literature [LRSW99, CL01, Lys02]. And it is still elusive when it comes to practical protocols: our solution is not efficient enough to be used in practice.

In conclusion, we need signatures of knowledge as a primitive because it comes up again and again in privacy-preserving protocols. This primitive is both conceptually helpful in understanding existing constructions (group signatures, ring signatures), and useful for developing new ones (signing without disclosing certification data, delegatable anonymous credentials).

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, multi-party 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;

[1] This is fictional; you do not need permission to drive on campus.

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 [Sho04, BR04]. 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. One 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 [DP92, SCP00] and a simulation-sound non-interactive zero-knowledge proof scheme with efficient

provers[Sah99, dSdCO+01] (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 [CS97], who use this term to mean a proof of knowledge (more specifically, a Σ-protocol [Cra97]) turned into a signature using the Fiat-Shamir heuristic [FS87]. 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 our protocols will work as desired, no

matter what larger system they may be operating within. We will begin by giving a UC definition of signatures of knowledge. For an overview of the UC framework, see Appendix A.

We begin by recalling Canetti's signature functionality. Note that the cited version of the functionality is from 2005, and is different from the one that Canetti first proposed in 2000. For details see Appendix B. For a detailed discussion and justification for Canetti's modelling choices see [Can05].

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.

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

Key Generation Upon receiving a value (`KeyGen`,sid) from some party P, verify that $sid = (P, sid')$ for some sid'. If not, then ignore the request. Else, hand (`KeyGen`,sid) to the adversary. Upon receiving (`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 (`VerificationAlgorithm`, sid, Verify) to P.

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

Signature Verification Upon receiving a value (`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 (Unforgeability error) to V and halt. Else, output (`Verified`,$sid, m, \mathsf{Verify}'(m, \sigma)$) to V.

As mentioned above, signatures of 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 Simsign and 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 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 an error message (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).

We are now ready to provide a more formal and concise description of the $\mathcal{F}_{SOK}(L)$ functionality.

$\mathcal{F}_{SOK}(L)$: 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_p.

Definition 2.1 (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 $\mathsf{Setup}, \mathsf{Sign}$, 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 $\mathcal{F}_{SOK}$ (with minor alterations). For more information, see Appendix 4.4.

2.2 A Definition in Terms of Games

We now give a second, games style definition for signatures of knowledge. We find that games style definitions are often more intuitive, particularly to a reader not thoroughly versed in the UC composability framework, and that they can also be much easier to work with. This definition provides additional clarity and also makes our job easier when proving security of our construction. 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 $\mathsf{Simsetup}$ that generates the public parameters (possibly from some different but indistinguishable distribution) together with the trapdoor, and $\mathsf{Simsign}$ which then signs using these public parameters. We require that no adversary can tell that he is interacting with $\mathsf{Simsetup}$ and $\mathsf{Simsign}$ 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 $\mathsf{Simsetup}$ and $\mathsf{Simsign}$ (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 $\mathsf{Simsetup}$ instead of Setup is because the extractor needs a trapdoor to be able to extract. Note that it also interacts with $\mathsf{Simsign}$ instead of Sign. The adversary could run Sign itself, so access to $\mathsf{Simsign}$ gives it a little bit of extra power.

Definition 2.2 (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 $(\mathsf{Setup}, \mathsf{Sign}, \mathsf{Verify})$ 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 $m \in Mes_k$

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

Simulatability There exists a polynomial time simulator consisting of algorithms $\mathsf{Simsetup}$ and $\mathsf{Simsign}$ 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 [2] In addition to ($\mathsf{Simsetup}, \mathsf{Simsign}$), there exists an extractor algorithm $\mathsf{Extract}$ 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 $s \in \{0,1\}^{f(k)}$

$$\begin{array}{rl} \Pr & [(p,\tau) \leftarrow \mathsf{Simsetup}(1^k); (M_L, x, m, \sigma) \leftarrow \mathcal{A}^{\mathsf{Sim}(p,\tau,\cdot,\cdot,\cdot,\cdot)}(s,p); \\ & \qquad w \leftarrow \mathsf{Extract}(p, \tau, M_L, x, m, \sigma) : \\ & M_L(x,w) \vee (M_L, x, m) \in Q^+ \vee \neg \mathsf{Verify}(p, M_L, x, m, \sigma)] \quad = 1 - \nu(k) \end{array}$$

where Q^+ denotes the query tape which lists all the previous successful queries $\mathcal{A}$ has sent to the oracle Sim., i.e. all those queries (M_L, x, m) which were sent with some valid witness w.

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 [GMR88]. 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_1 be the probability that the forgery is successful.

Adversary vs. Simulator: The simulator generates parameters using $\mathsf{Simsetup}$. 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_2 be the probability that the forgery is successful.

Adversary vs. Extractor: The extractor generates parameters using $\mathsf{Simsetup}$. 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 $\mathsf{Extract}$ to obtain a potential witness w. Let p_3 be the probability that w is a valid witness.

Adversary vs. Reduction: The reduction is given an instance x, which it publishes. It then generates parameters using $\mathsf{Simsetup}$. 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_4 be the probability that w is a valid witness.

By the simulatability property, the difference between p_1 and p_2 must be negligible. By the extraction property, the difference between p_2 and p_3 must be negligible. Since Sim ignores w and runs $\mathsf{Simsign}$, p_3 and p_4 must be identical. Thus, generating forgeries is at least as hard as deriving

[2] There was a typo in the original version.

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.

More formally, let G be an algorithm which samples (x, w) pairs, and X be an algorithm which samples x values from the same distribution. Let M_L be the language from which the x instances are chosen, and let s be any auxiliary input to the adversary.

Consider the following traditional signature scheme:

$\text{SigKeyGen}(1^k)$: $(x, w) \leftarrow G(1^k); p \leftarrow \mathsf{Setup}(1^k); PK = (x, p), SK = w$

$\text{SigSign}(m, SK)$: $\mathsf{Sign}(p, M_L, x, w, m)$

$\text{SigVerify}(m, \sigma, PK)$: $\mathsf{Verify}(p, M_L, x, m, \sigma)$

Theorem 2.1. *For all adversaries, the probability of forgery in the above scheme is at most negligibly higher than the probability of finding a valid withness w for a random $x \leftarrow X(1^k)$*

Proof. As defined above,

$$p_1 = \Pr\ [p \leftarrow \mathsf{Setup}(1^k); (x, w) \leftarrow G(1^k); (m, \sigma) \leftarrow A^{\mathsf{Sign}(p, M_L, x, w, \cdot)}(s, p, x) : (M_L, x, m) \notin Q^+ \wedge \mathsf{Verify}(p, M_L, x, m, \sigma)]$$

$$p_2 = \Pr\ [(p, \tau) \leftarrow \mathsf{Simsetup}(1^k); (x, w) \leftarrow G(1^k); (m, \sigma) \leftarrow A^{\mathsf{Sim}(p, \tau, M_L, x, w, \cdot)}(s, p, x) : (M_L, x, m) \notin Q^+ \wedge \mathsf{Verify}(p, M_L, x, m, \sigma)]$$

$$p_3 = \Pr\ [(p, \tau) \leftarrow \mathsf{Simsetup}(1^k); (x, w) \leftarrow G(1^k); (m, \sigma) \leftarrow A^{\mathsf{Sim}(p, \tau, M_L, x, w, \cdot)}(s, p, x); w' \leftarrow \mathsf{Extract}(p, \tau, x, M_L, m, \sigma) : (M_L, x, m) \notin Q^+ \wedge M_L(x, w')]$$

$$p_4 = \Pr\ [(p, \tau) \leftarrow \mathsf{Simsetup}(1^k); x \leftarrow X(1^k); (m, \sigma) \leftarrow A^{\mathsf{Simsign}(p, \tau, M_L, x, \cdot)}(s, p, x); w' \leftarrow \mathsf{Extract}(p, \tau, x, M_L, m, \sigma) : (M_L, x, m) \notin Q^+ \wedge M_L(x, w')]$$

Note that in the fourth scenario, success occurs if we find a valid witness for a random $x \leftarrow X(1^k)$. Thus, we have only to show that the probability of forgery (p_1) is a most negligibly higher than the probability of success in this scenario(p_4). We do this in several increments:

- Suppose there exists an adversary such that $p_1 - p_2$ is nonnegligible. Then we can show that the simulatability property does not hold for this signature of knowledge scheme. We construct a distinguisher D, which interacts with A and breaks the simulatability property.

  D receives p and s, generates a random (x, w) pair, and sends s, p, x to A. D answers A's signature queries ($\text{Sign}(p, M_L, x, w, \cdot)$) by forwarding them to his signing oracle. When A outputs a purported forgery (m, σ), D checks that m was not one of A's previous signing queries and that $\mathsf{Verify}(p, M_L, x, m, \sigma)$ returns true. If so, D returns 0; otherwise D returns 1.

  Now, note that

  $$\begin{aligned}
  &\Pr[p \leftarrow \mathsf{Setup}(1^k); b \leftarrow D^{\mathsf{Sign}(p, \cdot, \cdot, \cdot)}(s, p) : b = 1] \\
  &= \Pr[p \leftarrow \mathsf{Setup}(1^k); (x, w) \leftarrow G(1^k); (m, \sigma) \leftarrow A^{\mathsf{Sign}(p, M_L, x, w, \cdot)}(s, p, x) : (M_L, x, m) \notin Q^+ \wedge \mathsf{Verify}(p, M_L, x, m, \sigma)] \\
  &= p_1 \\
  &\Pr[(p, \tau) \leftarrow \mathsf{Simsetup}(1^k); b \leftarrow D^{\mathsf{Sim}(p, \tau, \cdot, \cdot, \cdot)}(s, p) : b = 1] \\
  &= \Pr[(p, \tau) \leftarrow \mathsf{Simsetup}(1^k); (x, w) \leftarrow G(1^k); (m, \sigma) \leftarrow A^{\mathsf{Sim}(p, \tau, M_L, x, w, \cdot)}(s, p, x) : (M_L, x, m) \notin Q^+ \wedge \mathsf{Verify}(p, M_L, x, m, \sigma)] \\
  &= p_2
  \end{aligned}$$

  By assumption, $p_1 - p_2$ is nonnegligible, so this contradicts the simulatability property.

- Suppose there exists an adversary A such that $p_2 - p_3$ is nonnegligible. Then we show that the extraction property does not hold. We construct an algorithm B which interacts with A and breaks the extraction property as follows.

  B receives p, s, generates a random (x, w) pair by running G and sends s, x, p to A. B answers A's signature queries by forwarding them to his signing oracle (Sim). When A outputs a potential forgery (m, σ), B outputs this signature pair. Let E be the process by which p, τ, x, w, m, w' and σ are chosen in the Adversary vs. Extractor and Adversary vs. Reduction games above. I.e. Let $E = \{(p, \tau) \leftarrow \mathsf{Simsetup}(1^k); (x, w) \leftarrow G(1^k); (m, \sigma) \leftarrow \mathcal{A}^{\mathsf{Sim}(p,\tau,M_L,x,w,\cdot)}(s, p, x); w' \leftarrow \mathsf{Extract}(p, \tau, M_L, x, m, \sigma)\}$ be a process which outputs (p, x, m, σ, w').

  Now we show that B breaks the extraction property:

$$
\begin{aligned}
&\Pr\quad [(p,\tau) \leftarrow \mathsf{Simsetup}(1^k); (M_L, x, m, \sigma) \leftarrow B^{\mathsf{Sim}(p,\tau,\cdot,\cdot,\cdot,\cdot)}(s,p);\\
&\qquad\qquad w' \leftarrow \mathsf{Extract}(p,\tau,M_L,x,m,\sigma) :\\
&\qquad \neg M_L(x,w') \wedge (M_L,x,m) \notin Q^+ \wedge \mathsf{Verify}(p, M_L, x, m, \sigma)]\\
&= \Pr\quad [(p,\tau) \leftarrow \mathsf{Simsetup}(1^k); (x,w) \leftarrow G(1^k); (m,\sigma) \leftarrow \mathcal{A}^{\mathsf{Sim}(p,\tau,M_L,x,w,\cdot)}(s,p,x);\\
&\qquad\qquad w' \leftarrow \mathsf{Extract}(p,\tau,M_L,x,m,\sigma) :\\
&\qquad \neg M_L(x,w') \wedge (M_L,x,m) \notin Q^+ \wedge \mathsf{Verify}(p, M_L, x, m, \sigma)]\\
&= \Pr[(p,x,m,\sigma,w') \leftarrow E(1^k) : \neg M_L(x,w') \wedge (M_L,x,m) \notin Q^+ \wedge \mathsf{Verify}(p, M_L, x, m, \sigma)]\\
&> \Pr[(p,x,m,\sigma,w') \leftarrow E(1^k) : \neg M_L(x,w') \wedge (M_L,x,m) \notin Q^+ \wedge \mathsf{Verify}(p, M_L, x, m, \sigma)]\\
&\quad - \Pr[(p,x,m,\sigma,w') \leftarrow E(1^k) : M_L(x,w') \wedge (M_L,x,m) \notin Q^+ \wedge \neg\mathsf{Verify}(p, M_L, x, m, \sigma)]\\
&= \Pr[(p,x,m,\sigma,w') \leftarrow E(1^k) : \neg M_L(x,w') \wedge (M_L,x,m) \notin Q^+ \wedge \mathsf{Verify}(p, M_L, x, m, \sigma)]\\
&\quad + \Pr[(p,x,m,\sigma,w') \leftarrow E(1^k) : M_L(x,w') \wedge (M_L,x,m) \notin Q^+ \wedge \mathsf{Verify}(p, M_L, x, m, \sigma)]\\
&\quad - \Pr[(p,x,m,\sigma,w') \leftarrow E(1^k) : M_L(x,w') \wedge (M_L,x,m) \notin Q^+ \wedge \mathsf{Verify}(p, M_L, x, m, \sigma)]\\
&\quad - \Pr[(p,x,m,\sigma,w') \leftarrow E(1^k) : M_L(x,w') \wedge (M_L,x,m) \notin Q^+ \wedge \neg\mathsf{Verify}(p, M_L, x, m, \sigma)]\\
&= \Pr[(p,x,m,\sigma,w') \leftarrow E(1^k) : (M_L,x,m) \notin Q^+ \wedge \mathsf{Verify}(p, M_L, x, m, \sigma)]\\
&\quad - \Pr[(p,x,m,\sigma,w') \leftarrow E(1^k) : M_L(x,w') \wedge (M_L,x,m) \notin Q^+]\\
&= p_2 - p_3 \quad > \epsilon(k) \text{ for some nonnegligible function } \epsilon(k).
\end{aligned}
$$

  Thus,

$$
\begin{aligned}
&\Pr\quad [(p,\tau) \leftarrow \mathsf{Simsetup}(1^k); (M_L, x, m, \sigma) \leftarrow B^{\mathsf{Sim}(p,\tau,\cdot,\cdot,\cdot,\cdot)}(s,p);\\
&\qquad\qquad w \leftarrow \mathsf{Extract}(p,\tau,M_L,x,m,\sigma) :\\
&\qquad M_L(x,w) \vee (M_L,x,m) \in Q^+ \vee \neg\mathsf{Verify}(p, M_L, x, m, \sigma)] \quad < 1 - \epsilon(k)
\end{aligned}
$$

  which breaks the extraction property.

- As noted above, since X draws x values from the same distribution as G and since Sim simply ignores w and performs $\mathsf{Simsign}$, we must have $p_3 = p_4$.

Thus, $p_1 - p_4$ is negligible. That means, the probability of forgery in Alice's signature scheme is at most negiligibly higher than the probability of finding a witness w for a random instance $x \leftarrow X$. □

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 [Can01]. 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 (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 (CRS, sid, v) to P and also sends (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 (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 (CRS,v), and stores v. Finally, it returns (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 (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 (Signature, $sid, m, x, \mathsf{Sign}(v, M_L, x, w, m)$). When P receives a request (Verify, sid, m, x, σ) from the environment, it again retrieves the stored v, and then returns (Verified, $sid, m, x, \sigma, \mathsf{Verify}(v, M_L, x, m, \sigma)$).

Recall that U_p, defined in Definition 2.1, is the universal language.

Theorem 2.2. *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. 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 (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 (Algorithms,sid, Sign, Verify, Simsign, Extract) back to $\mathcal{F}_{SOK}$. Another place where S must do something is when the adversary A queries the $\mathcal{F}_{CRS}^{D}$ functionality. In response to such a query, 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 Completeness error or Unforgeability error.

Suppose that the probability of a completeness error is non-negligible. By construction of S, a completeness error happens when a signature generated by Simsign is rejected by Verify. This contradicts the simulatability requirement: Since the environment Z can, with non-negligible probability, find a series of queries to $\mathcal{F}_{SOK}$ that lead to a completeness error, then it distinguishes the output of Sim as defined in definition 2.2 from that of Sign, since by SimExt-security of Σ we get the property that signatures generated by Sign are always accepted by Verify.

Suppose that the probability of an unforgeability error is non-negligible. Then there exists some polynomial $i(k)$, such that the probability that the $i(k)^{th}$ verification query causes the error, is non-negligible. By construction of S, an unforgeability error happens when Extract fails to

extract a witness w from a signature σ, issued by Z, that is accepted by Verify but which was not generated by $\mathcal{F}_{SOK}$. Let us construct an adversary $\mathcal{A}$ that uses such Z to break the unforgeability property of Σ. By definition of SimExt-security, first $(p, \tau) \leftarrow \mathsf{Simsetup}(1^k)$ are generated, and then $\mathcal{A}^{\mathsf{Sim}(p,\tau,\cdot,\cdot,\cdot,\cdot)}(s, p)$ is invoked, where s is an auxiliary string, for example one that contains the description of Z. $\mathcal{A}$ then invokes Z. When Z issues setup queries ($\mathtt{Setup}$,sid), $\mathcal{A}$ returns the appropriate $\mathsf{Sign}(p, \cdot, \cdot, \cdot)$ and $\mathsf{Verify}(p, \cdot, \cdot, \cdot)$ algorithms. When Z issues signing queries, $\mathcal{A}$ uses its Sim oracle; note that by construction of S, the resulting responses are distributed identically to the responses that $\mathcal{F}_{SOK}$ would have issued. For the first $i(k)-1$ verification queries ($\mathtt{Verify}$,sid, m, x, σ of Z, $\mathcal{A}$ returns the output of $\mathsf{Verify}(p, m, x, \sigma)$. $\mathcal{A}$ outputs the contents of Z's $i(k)^{th}$ verification query (m, x, σ). Note that Z's view here is the same as its view in an interaction with $\mathcal{F}_{SOK}$, and so the probability that $\mathcal{F}_{SOK}$ would halt with an unforgeability error is non-negligible. Unforgeability errors occur when $\mathsf{Extract}(p, \tau, m, x, \sigma)$ fails to extract a valid witness w, and m has not been signed by $\mathcal{F}_{SOK}$, and yet $\mathsf{Verify}(p, m, x, \sigma) = Accept$. Thus, with non-neglible probability, $\mathcal{A}$ produces (m, x, σ) that violate the conditions of the extractability property of the SimExt-security definition. Therefore, Σ is not SimExt-secure, which is a contradiction.

Therefore we have shown that the probability of event E is negligible. Let us now show that, 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. Suppose for contradiction that it is distinguishable. Then let us use Z to construct a distinguisher $\mathcal{A}$ that will contradict the simulatability property of Σ, namely, distinguish between the Sign oracle and the Sim oracle. $\mathcal{A}$ will invoke Z. It will respond to Z's setup query by giving it $\mathsf{Sign}(p, \cdot, \cdot, \cdot)$ and $\mathsf{Verify}(p, \cdot, \cdot, \cdot)$. If p was generated using Setup (and so $\mathcal{A}$ is given Sign as its oracle), this is the same situation as when Z is interacting with π_Σ; while if it was generated using $\mathsf{Simsetup}$ (and so $\mathcal{A}$'s oracle is Sim), then this is the same as when Z is interacting with S and $\mathcal{F}_{SOK}$. In response to Z's signing queries, $\mathcal{A}$ will ask its oracle to produce a signature. Again, note that if $\mathcal{A}$'s oracle is Sign, σ is distributed as in π_Σ, while if it is Sim, σ is distributed as in an interaction with $\mathcal{F}_{SOK}$— the only other possibility for $\mathcal{F}_{SOK}$ would be to halt with an error, but we are considering the case when this does not happen. To respond to Z's verification queries, $\mathcal{A}$ runs the Verify algorithm. Since we have conditioned on $\mathcal{F}_{SOK}$ not halting with an error, the response of the Verify algorithm will, in case $\mathcal{A}$'s oracle is Sim, correspond to the behavior of $\mathcal{F}_{SOK}$. On the other hand, if $\mathcal{A}$'s oracle is Sign, this response is the same as in π_Σ. Therefore, if Z distinguishes $\mathcal{F}_{SOK}$ and S from π_Σ and A, it implies that $\mathcal{A}$ distinguishes between the two oracles, which contradicts simulatability.

Now let us show the other direction. Suppose that π_Σ UC-realizes $\mathcal{F}_{SOK}(U)$ in the $\mathcal{F}_{CRS}^D$-hybrid model. Let us show that it follows that Σ is SimExt-secure. Since π_Σ is UC-realizable, it must have a simulator S. By $(p, \tau) \leftarrow \mathsf{Simsetup}(1^k)$ let us refer to the algorithm that S runs in response to a setup query; the public parameters p consist of the value that S will subsequently return in response to queries to the $\mathcal{F}_{CRS}^D$ functionality; the trapdoor τ consists of the algorithms $(\mathsf{Simsign}, \mathsf{Extract})$ that S hands over to $\mathcal{F}_{SOK}$ in response to the setup query.

Suppose Σ does not satisfy the correctness property and yet π_Σ UC-realizes $\mathcal{F}_{SOK}(U)$. Then let us show a contradiction. Note that honest parties that use $\mathcal{F}_{SOK}$, always accept signatures generated by honest parties through $\mathcal{F}_{SOK}$, (although $\mathcal{F}_{SOK}$ may halt during signing with a $\mathsf{Correctness\ error}$). On the other hand, since Σ does not satisfy completeness, honest parties using Σ reject signatures produced by honest parties that use Σ with non-negligible probability. Therefore, to distinguish $\mathcal{F}_{SOK}$ from π_Σ, all Z needs to do is to find a signature σ, output by an honest party, that is rejected by Verify. Such a signature will exist in π_Σ with non-negligible probability since

Σ does not satisfy correctness, and yet will not exist in $\mathcal{F}_{SOK}$ by the argument above. This is a contradiction.

Suppose Σ does not satisfy simulatability. Then there exists a distinguisher between the Sign and the Sim oracles. Then it is easy to see how the environment can use such a distinguisher to distinguish between $\mathcal{F}_{SOK}$ and π_Σ, since in $\mathcal{F}_{SOK}$ signatures output by honest parties are computed according to Sim, while in π_Σ, they are computed according to Sign.

Finally, suppose Σ does not satisfy extractability. Then there exists an adversary $\mathcal{A}$ that, with non-negligible probability produces a signature from which an appropriate witness cannot be extracted. Then it is easy to see how the environment Z can use this adversary to distinguish π_Σ from $\mathcal{F}_{SOK}$. It will invoke $\mathcal{A}$; whenever $\mathcal{A}$ issues queries to Sim, it will direct these queries to be signed by honest parties. Finally, $\mathcal{A}$ produces an unsigned message m and (m, x, σ) that Verify accepts, but from which a witness cannot be extracted. Z directs (m, x, σ) to be verified by an honest party. If $\mathcal{A}$'s success happens non-negligibly often, then, should Z be talking to $\mathcal{F}_{SOK}$, it will cause $\mathcal{F}_{SOK}$ to halt with an error with non-negligible probability; while should it be talking to π_Σ, the values (m, x, σ) will be accepted.

□

3 Construction

Here we present Σ, a construction of a SimExt-secure signature of knowledge. By Theorem 2.2, 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 [DP92, SCP00] and simulation-sound non-interactive zero knowledge proofs [Sah99, dSdCO+01]. (For a review of these primitives, see Appendix C.) 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.

Theorem 3.1. *The construction above is a SimExt-secure signature of knowledge.*

Proof. (Sketch) First we argue simulatability. In the 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 NIZKSimsetup. In the Simsign phase, the simulator will always let c be the encryption of $0^{|m|+l_L}$, and will create (fake) proof π by invoking NIZKSim.

We show that the resulting simulation is successful using a two-tier hybrid argument. First, note that, by the unbounded zero-knowledge property of the underlying NIZK proof system, signatures obtained by replacing calls to NIZKProve by calls to NIZKSim will be distributed indistinguishably from real signatures. We call this signing process MixSign; so we see that MixSign is indistinguishable from Sign. Second, note that, 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. Since the only difference between MixSign and Simsign is in how c is chosen, it follows that MixSign and Simsign are indistinguishable as well. So we get simulatability.

Second, let us argue extraction. Recall that, as part of the trapdoor τ, 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. □

Proof. (Full)

- Simulatability:

  Simsetup runs $(PK, SK) \leftarrow \mathcal{G}(1^k)$ to obtain a random public key PK and the corresponding secret key SK. Then, let $(\rho, \tau') \leftarrow \mathsf{NIZKSimsetup}(1^k)$. Let $p = PK \circ \rho$, $\tau = (SK, \tau')$.

  $\mathsf{Simsign}(p, \tau, M_L, x, m)$ parses $p = PK \circ \rho$ and $\tau = (SK, \tau')$, lets $c \leftarrow \mathsf{Enc}(PK, (m, 0^{l_L}))$ and obtains $\pi = \mathsf{NIZKSim}(\rho, \tau', (m, M_L, x, c, PK), (\exists(w, R) \ : \ c = \mathsf{Enc}(PK, m, w, R) \wedge M_L(x, w)))$. Output (c, π).

  We will show that interaction with Sign and Setup is indistinguishable from interaction with Simsign and Simsetup. Consider an intermediate signing algorithm:

  $\mathsf{MixSign}(p, \tau, L, x, m, w)$ verifies that $M_L(x, w)$ accepts, parses $p = PK \circ \rho$ and $\tau = (SK, \tau')$, lets $c \leftarrow \mathsf{Enc}(PK, (m, w))$ and obtains $\pi = \mathsf{NIZKSim}(\rho, \tau', (m, M_L, x, c, PK), (\exists(w, R) \ : \ c = \mathsf{Enc}(PK, m, w, R) \wedge M_L(x, w)))$. Output (c, π)

  Let

  $$p_1 = \left| \begin{array}{l} \Pr[(p, \tau) \leftarrow \mathsf{Simsetup}(1^k); b \leftarrow \mathcal{A}^{\mathsf{MixSign}(p, \tau, \cdot, \cdot, \cdot, \cdot)}(s, p) \ : \ b = 1] \\ - \ \Pr[p \leftarrow \mathsf{Setup}(1^k); b \leftarrow \mathcal{A}^{\mathsf{Sign}(p, \cdot, \cdot, \cdot, \cdot)}(s, p) \ : \ b = 1] \end{array} \right|$$

  Sign and MixSign are identical except that Sign makes calls to NIZKProve, and MixSign makes calls to NIZKSim. Thus, if p_1 is nonnegligible, then we have broken the Unbounded Zero Knowledge property of the SSNIZK proof system.

  Finally, consider the following hybrid signing algorithms:

  $\mathsf{HybridSign}_i$ calls MixSign the first i times it is queried, and calls Sim for the rest of the queries.

Note that $\mathsf{HybridSign}_i$ and $\mathsf{HybridSign}_{i+1}$ are identical except that on the $(i+1)$-th call, $\mathsf{HybridSign}_{i+1}$ encrypts (m_{i+1}, w_{i+1}), while $\mathsf{HybridSign}_i$ encrypts $(m_{i+1}, 0^{l_L})$.

That means if

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

is nonnegligible, then we have broken the semantic security of the encryption scheme.

Thus, since $\mathsf{HybridSign}_0 = \mathsf{Simsign}$, and $\mathsf{HybridSign}_q = \mathsf{MixSign}$ where q is the total(polynomial) number of signing queries $\mathcal{A}$ makes, we now have

$$p_2 = \left| \begin{array}{l} \Pr[(p,\tau) \leftarrow \mathsf{Simsetup}(1^k); b \leftarrow \mathcal{A}^{\mathsf{Sim}(p,\tau,\cdot,\cdot,\cdot,\cdot)}(s,p) \ : \ b=1] \\ \ - \ \Pr[(p,\tau) \leftarrow \mathsf{Simsetup}(1^k); b \leftarrow \mathcal{A}^{\mathsf{MixSign}(p,\tau,\cdot,\cdot,\cdot,\cdot)}(s,p) \ : \ b=1] \end{array} \right|$$

is negligible.

So finally, combining p_1 and p_2 , we have that

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

is negligible. Thus, we have shown Simulatability.

- Extraction:

  $\mathsf{Extract}(p, \tau, M_L, x, m, \sigma)$ parses $\sigma = (c, \pi)$ and $\tau = (SK, \tau')$, runs $\mathsf{Dec}(PK, SK, c)$ to obtain (m, w), and outputs w.

  Suppose there exists $\mathcal{A}$, f, s such that

$$\begin{array}{rl} \Pr & [(p,\tau) \leftarrow \mathsf{Simsetup}(1^k); (M_L, x, m, \sigma) \leftarrow \mathcal{A}^{\mathsf{Sim}(p,\tau,\cdot,\cdot,\cdot,\cdot)}(s,p); \\ & \qquad\qquad w \leftarrow \mathsf{Extract}(p, \tau, M_L, x, m, \sigma) : \\ & M_L(x,w) \vee (M_L, x, m) \in Q^+ \vee \neg \mathsf{Verify}(p, M_L, x, m, \sigma)] \quad = 1 - \epsilon(k) \end{array}$$

  for nonnegligible $\epsilon(k)$

  Let L' be a language such that an instance $(\rho, x, M_L, m, c, PK) \in L'$ iff there exists (w, R) such that $M_L(x, w) \wedge c = \mathsf{Enc}(PK, (m, w), R)$.

  Then, from above, using the constructions for Simsetup, Extract, and Verify:

$$\begin{array}{rl} \Pr & [(PK \circ \rho, SK \circ \tau) \leftarrow \mathsf{Simsetup}(1^k); (M_L, x, m, \sigma = (c, \pi)) \leftarrow \mathcal{A}^{\mathsf{Sim}(p,\tau,\cdot,\cdot,\cdot,\cdot)}(s,p); \\ & \qquad\qquad (m, w) = \mathsf{Dec}(PK, SK, c) : \\ & M_L(x,w) \vee (M_L, x, m) \in Q^+ \vee \neg(\mathsf{NIZKVerify}(\pi, (\rho, x, M_L, m, c, PK) \in L'))] \quad = 1 - \epsilon(k) \end{array}$$

  Looking at the probability of the opposite event, we get:

$$\begin{array}{rl} \Pr & [(PK \circ \rho, SK \circ \tau) \leftarrow \mathsf{Simsetup}(1^k); (M_L, x, m, \sigma = (c, \pi)) \leftarrow \mathcal{A}^{\mathsf{Sim}(p,\tau,\cdot,\cdot,\cdot,\cdot)}(s,p); \\ & \qquad\qquad (m, w) = \mathsf{Dec}(PK, SK, c) : \\ & \qquad (M_L, x, m) \notin Q^+ \wedge \mathsf{NIZKVerify}(\pi, (\rho, x, M_L, m, c, PK) \in L') \\ & \qquad\qquad \wedge \neg M_L(x, w)] \quad = \epsilon(k) \end{array}$$

Finally, using the definition of L', which says that when w is obtained from $\mathsf{Dec}(PK, SK, c)$, $M_L(x, w)$ must accept:

$$\begin{aligned}\Pr \quad [(PK \circ \rho, SK \circ \tau) \leftarrow \mathsf{Simsetup}(1^k); (M_L, x, m, \sigma = (c, \pi)) \leftarrow \mathcal{A}^{\mathsf{Sim}(p,\tau,\cdot,\cdot,\cdot,\cdot)}(s, p); \\ (m, w) = \mathsf{Dec}(PK, SK, c) : \\ (M_L, x, m) \notin Q^+ \wedge \mathsf{NIZKVerify}(\pi, (\rho, x, M_L, m, c, PK) \in L') \\ \wedge (\rho, x, M_L, m, x, PK) \notin L'] \quad = \epsilon(k)\end{aligned}$$

Since Sim calls $\mathsf{NIZKSim}$ for each of the queried proofs and we now have an algorithm which can prove false statements, we have broken the simulation soundness property.

□

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 $c = Enc(PK_E, p; R)$, 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 key x and maps it to her public key 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 resulting 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, in other words 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 4.1: 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.

Then we generalize the idea to apply to languages L that are not defined by just one functionality $\mathcal{F}_0$, but by a set of functionalities $\mathcal{F}_1, \ldots, F_c$. For example, it will follow that we can define and UC-realize our ideal functionality for group signatures, where the underlying languages is defined in terms of $\mathcal{F}_f$, $\mathcal{F}_{Enc}$, and F_Σ. This extension is presented in Section 4.3. Further, we give a multiple-session extension of $\mathcal{F}_{SOK}$ that allows protocols to reuse the CRS; this is presented in Section 4.4. In this multiple-session extension, we can even allow multiple signature of knowledge instances to sign on behalf of languages defined in terms of the same subfunctionality instances.

In addition, it will follow that $\mathcal{F}_{SOK}(\ldots(\mathcal{F}_{SOK}(\mathcal{F}_1, \ldots, \mathcal{F}_t))\ldots)$ is UC-realizable, and so a signature of knowledge can serve as a witness for another signature of knowledge. This allows us to UC-realize delegatable anonymous credentials. This is explained in Section 4.5.

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.) As a result, our modelling task at hand is very complex. For simplicity, we will only formally address the situation of $\mathcal{F}_{SOK}(\mathcal{F})$, i.e., when the language L is defined by only one sub-functionality $\mathcal{F}$.

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 4.1 (Verification functionality)**.** A functionality $\mathcal{F}$ is a verification functionality if (1) there exists some $\mathit{start}(\mathcal{F})$ 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 $\mathsf{Verify}(\mathit{input}, \mathit{witness})$ or halts with an error.

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

$start(\mathcal{F})$ =Setup.

Any verification functionality $\mathcal{F}$ with a particular *sid* defines a language of inputs that will be accepted by this functionality if an appropriate witness is provided; moreover, this language can be captured by a deterministic polynomial-time Turing machine represented by the Verify algorithm. The only times when Verify's behavior is different from that of $\mathcal{F}$ cause $\mathcal{F}$ to halt with an error. To work with this language, we need a way to obtain the Turing machine Verify. Note that Canetti's signature functionality does not make the algorithm Verify freely available to any party that calls it. However, it could easily be extended to make Verify explicitly available through an extra query:

Definition 4.2 (Explicit verification functionality)**.** Let $\mathcal{F}$ be a verification functionality. It is also 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 polynomial time algorithm Verify.

An explicit verification functionality not only defines a language L, but also makes available the Turing machine M_L for 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 here the Turing machine 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. (Sign, Verify, Simsign, 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 that just because $M_L(x, w)$ accepts, does not mean that $\mathcal{F}_0$'s verification procedure necessarily accepts. However, $\mathcal{F}_{SOK}$ 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 (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 (more about that later).

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.

$\mathcal{F}_{SOK}(\mathcal{F}_0)$: 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 queried 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 an 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 ($\mathcal{F}_0$-$command, sid, sid_0, data$). Upon receiving a message 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.

Theorem 4.1. *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 $\mathcal{F}_{CRS}^D$ hybrid model iff $\mathcal{F}_0$ is nontrivially UC-realizable in the $\mathcal{F}_{CRS}^D$ hybrid model, where we consider a realization to be nontrivial if it never halts with an error message.*

Proof. Assume that there exists a SimExt secure signature of knowledge scheme (Setup, Sign, Verify). Assume also that there exists a protocol ρ which UC-realizes $\mathcal{F}_0$. Then we build a protocol π in the $\mathcal{F}_{CRS}^D$ hybrid model which UC-realizes $\mathcal{F}_{SOK}(\mathcal{F}_0)$ as follows.

Upon receiving a value (Setup, $sid = (\mathcal{F}_0, sid_0, sid_1)$), if this is the first such message, the protocol π will start running an instance of ρ. It will send ρ inputs (Setup, sid_0) and then

(`VerificationAlgorithm`, sid_0), and ρ will return (`VerificationAlgorithm`, $sid_0, \mathcal{F}_0 M$). π will store $M_L = \mathcal{F}_0 M$. From this point on, it will behave as π_Σ defined in Section 2.2. To finish the Setup query, it will obtain the CRS p from $\mathcal{F}_{CRS}^D$ (where D is the distribution of $\mathsf{Setup}(1^k)$) and output the $\mathsf{Sign}(p, M_L, \cdot, \cdot, \cdot)$ and $\mathsf{Verify}(p, M_L, \cdot, \cdot, \cdot)$ algorithms.

On Sign and Verify queries, it will behave exactly as π_Σ defined in Section 2.3.

When π receives a query of the form ($\mathcal{F}_0$-*query*, $sid_0, sid_1, data$) from party P, it will send ($query, sid_0, data$) to ρ on behalf of party P. When it receives a response, ($response, sid_0, data$) from ρ, it forwards ($\mathcal{F}_0$-*response*, $sid_0, data$) to P.

Similarly, when ρ wants to send ($command, sid_0, data$) to the adversary, π will give to the adversary the message ($\mathcal{F}_0$-*command*, $sid, sid_0, data$). When π receives a message ($\mathcal{F}_0$-*header*, $sid, sid_0, data$) from the adversary, it will give ($header, sid_0, data$) to ρ on behalf of the adversary.

On input (`VerificationAlgorithm`,sid) from party P, π will output (`VerificationAlgorithm`,sid, $\mathsf{Verify}(v, M_L, \cdot, \cdot, \cdot)$) to P.

We will show that π UC-realizes $\mathcal{F}_{SOK}(\mathcal{F}_0)$.

Consider the hybrid protocol H, which runs in the $\mathcal{F}_0$-hybrid model. H will behave like π except that calls to ρ will be sent instead to the ideal $\mathcal{F}_0$ functionality. By the UC theorem, if H UC-realizes $\mathcal{F}_{SOK}(\mathcal{F}_0)$ in the $\mathcal{F}_0$-hybrid mode, then π UC-realizes $\mathcal{F}_{SOK}(\mathcal{F}_0)$. Thus, we have only to show that H UC-realizes $\mathcal{F}_{SOK}(\mathcal{F}_0)$ in the hybrid model.

Let us examine the definition of the $\mathcal{F}_{SOK}(\mathcal{F}_0)$ functionality more closely. Note that, conditioned on error with $\mathcal{F}_0$ never happening, we can equivalently define $\mathcal{F}_{SOK}(\mathcal{F}_0)$ as follows: in the Setup query, first obtain M_L using **Get Language** as a subroutine; then obtain the algorithms Sign and Verify by invoking an instance of $\mathcal{F}_{SOK}(U_p)$ with session id sid as a subroutine, where p is a polynomial bounding the running time of M_L, U_p is the universal language, and every instance x passed to $\mathcal{F}_{SOK}(U_p)$ is prepended by a description of M_L. (Recall that $\mathcal{F}_{SOK}(U_p)$ responds to its first valid (`Setup` sid) query by sending sid to the adversary, receiving and storing Sign, Verify, Simsign, and Extract algorithms, and returning Sign and Verify.) Similarly, to process Sign and Verify queries, compute the output of the query by making a call to the same instance of $\mathcal{F}_{SOK}(U_p)$, again with M_L prepended to the instance x. Let us denote this functionality by G. (The fact that G is an equivalent formulation, assuming "Error with $\mathcal{F}_0$" never occurs, follows by inspection of $\mathcal{F}_{SOK}(\mathcal{F}_0)$ and we will leave it to the reader to verify that fact.)

Note that if, in G, we replace all calls to $\mathcal{F}_{SOK}(U_p)$ by calls to π_Σ, we obtain H. Moreover, by Theorem 2.2, π_Σ realizes $\mathcal{F}_{SOK}(U_p)$, and therefore by the UC theorem, no environment Z can distinguish an execution with H from an execution with G.

Thus, we have shown that as long as $\mathcal{F}_{SOK}(\mathcal{F}_0)$ does not halt with an "Error with $\mathcal{F}_0$" error, $\mathcal{F}_{SOK}(\mathcal{F}_0)$ and H are indistinguishable, since in this case G is just another formulation of $\mathcal{F}_{SOK}(\mathcal{F}_0)$. By definition, since $\mathcal{F}_0$ is an explicit verification functionality, the call (`VerificationAlgorithm`, sid_0) to $\mathcal{F}_0$ returns $M_L = \mathsf{Verify}_{\mathcal{F}_0}$, the algorithm that $\mathcal{F}_0$ uses on its verification queries. Thus, $\mathcal{M}_L(x, w) = \mathsf{Verify}_{\mathcal{F}_0}(x, w) = 1$ iff $\mathcal{F}_0$ accepts on query (`Verify`, sid_0, x, w).

That means "Error with $\mathcal{F}_0$" can only occur if $\mathcal{F}_0$ halts with an error. However, we require that ρ which realizes $\mathcal{F}_0$ never outputs an error. Now suppose there exists an environment Z which can cause $\mathcal{F}_{SOK}(\mathcal{F}_0)$ to halt with "Error with $\mathcal{F}_0$" with nonnegligible probablility. Then we can simulate $\mathcal{F}_{SOK}$ and use Z to distinguish ρ from $\mathcal{F}_0$, since ρ never outputs an error, but $\mathcal{F}_0$ does.

Thus, with all but negligible probability, interaction with $\mathcal{F}_{SOK}(\mathcal{F}_0)$ and S is indistinguishable from interaction with G running with S, which is in turn indistinguishable from interaction with H in the $\mathcal{F}_0$-hybrid model. As stated above this implies that π UC-realizes $\mathcal{F}_{SOK}(\mathcal{F}_0)$.

Now we will consider the reverse direction. Assume that $\mathcal{F}_{SOK}(\mathcal{F}_0)$ is UC-realized by some protocol π in the $\mathcal{F}_{CRS}^D$ hybrid model. Then we will show that $\mathcal{F}_0$ is also UC-realizable.

Let ρ be the following protocol in the $\mathcal{F}_{SOK}(\mathcal{F}_0)$-hybrid model: On input $(start(\mathcal{F}_0)sid_0)$, ρ sets $sid = (\mathcal{F}_0, sid_0, sid_1)$ for some random sid_1, and sends (`Setup`,sid) to $\mathcal{F}_{SOK}(\mathcal{F}_0)$. On receiving any other query $(query, sid_0, data)$, ρ sends $(\mathcal{F}_0\text{-}query, sid_0, sid_1, data)$ to $\mathcal{F}_{SOK}(\mathcal{F}_0)$. When $\mathcal{F}_{SOK}(\mathcal{F}_0)$ sends a response $(\mathcal{F}_0\text{-}response, sid_0, data)$, ρ will output $(response, sid_0, data)$. When ρ receives a message $(\mathcal{F}_0\text{-}command, sid, sid_0, data)$ from $\mathcal{F}_{SOK}(\mathcal{F}_0)$, it sends $(command, sid_0, data)$ to the adversary. When the adversary sends a response $(header, sid_0, data)$, it sends $(\mathcal{F}_0\text{-}header, sid, sid_0, data)$ to $\mathcal{F}_{SOK}(\mathcal{F}_0)$.

Clearly, by definition of $\mathcal{F}_{SOK}(\mathcal{F}_0)$, ρ UC-realizes $\mathcal{F}_0$ in the $\mathcal{F}_{SOK}(\mathcal{F}_0)$-hybrid model. Thus, by the UC theorem, ρ^π(the protocol ρ where all calls to $\mathcal{F}_{SOK}(\mathcal{F}_0)$ have been replaced by calls to π) UC-realizes $\mathcal{F}_0$. Furthermore, since π nontrivially realizes $\mathcal{F}_{SOK}(\mathcal{F}_0)$, and ρ doesn't output any error messages, and so ρ^π nontrivially realizes $\mathcal{F}_0$.

□

4.3 Multiple Subfunctionalities

We defined $\mathcal{F}_{SOK}(\mathcal{F}_0)$ in terms of just one instance of a sub-functionality $\mathcal{F}_0$. It is easy to extend the ideal functionality to a more complex case. For example, suppose that $\mathcal{F}_1$ and $\mathcal{F}_2$ are both explicit verification functionalities, and we want to realize a signature of knowledge of either a an accepting input to $\mathcal{F}_1$ or accepting input to $\mathcal{F}_2$. In order to decide whether a given $(input, witness)$ pair is acceptable by $\mathcal{F}_1$ or $\mathcal{F}_2$, it is sufficient to query both functionalities and combine the result using the OR function. Similarly, given the Turing machines M_{L_1} and M_{L_2} for their respective verification algorithms, it is straightforward to produce a Turing machine for the verification algorithm of this extended functionality: just output a Turing machine that outputs the OR of M_{L_1} and M_{L_2}.

In general, if $\mathcal{F}_1, \ldots, \mathcal{F}_c$ are explicit verification functionalities, $f_1, \ldots, f_c$ and $f'_1, \ldots, f'_c$ are any poly-time computable functions, and b is any polynomial-time computable Boolean function, then we can specify a signature of knowledge functionality $\mathcal{F}_{SOK}(\mathcal{F}_1, \ldots, \mathcal{F}_c, f_1, f'_1, \ldots, f_c, f'_c, b)$ for signatures of knowledge of w such that $b(x, w, \mathcal{F}_1(f_1(x, w), f'_1(x, w)), \ldots, \mathcal{F}_c(f_c(x, w), f'_c(x, w))) = 1$. What $\mathcal{F}_{SOK}$ will need to do to decide whether w is a witness to x's membership in the language defined in terms of $\mathcal{F}_1, \ldots, \mathcal{F}_c$, $f_1, \ldots, f_c$ and $f'_1, \ldots, f'_c$ and the Boolean function b, is to: (1) store the response r_i of each $\mathcal{F}_i$ to a $(\mathsf{Verify}, sid, f_i(x, w), f'_i(x, w))$ query; (2) evaluate $b(x, w, r_1, \ldots, r_c)$. And again, given a TM for each M_L, we can easily build a TM for this composite language.

For example, this captures signatures of knowledge of a signature on one of several messages; on all of several messages; on a preimage of a one-way function; on a decryption of a ciphertext formed using a given string as randomness; etc. It is easy to see that this covers the group signature and ring signature applications we have discussed previously.

Note that for all such extensions, we can generalize the proof of Theorem 4.1 and show that, assuming SimExtsecure signatures of knowledge exist, the resulting $\mathcal{F}_{SOK}(\mathcal{F}_1, \ldots, \mathcal{F}_c, f_1, f'_1, \ldots, f_c, f'_c, b)$ is UC-realizable in the $\mathcal{F}_{CRS}^D$ hybrid model if and only if $\mathcal{F}_1, \ldots, \mathcal{F}_c$ are UC-realizable in the $\mathcal{F}_{CRS}^D$ hybrid model.

For another example of how $\mathcal{F}_{SOK}$ might use multiple subfunctionalities, see Section 4.5.

4.4 Multi-Session Extensions

Recall the definition of a multi-session extension $\hat{\mathcal{F}}$ of an ideal functionality due to Canetti and Rabin [CR03]. In a nutshell, a multi-session extension $\hat{\mathcal{F}}$ takes care of managing many instances of $\mathcal{F}$ in such a way that they don't interfere with each other. Specifically, when invoked $\hat{\mathcal{F}}$ is given a session id *sid*, and also a sub-session id *ssid*, and the query itself. $\hat{\mathcal{F}}$ creates an instance of $\mathcal{F}$ with sub-session id *ssid* if it does not already exist, and forwards the query to it. Thus, many instances of $\mathcal{F}$ run within $\hat{\mathcal{F}}$ as if they were running completely independently. Canetti and Rabin proved that for any protocol π, if a protocol ρ UC-realizes $\hat{\mathcal{F}}$, and protocol π UC-realizes a functionality G in the $\mathcal{F}$-hybrid model (with calls to many independent instances of $\mathcal{F}$), then $\pi^{[\rho]}$ UC-realizes G, where $\pi^{[\rho]}$ is the protocol π where all calls to instances of $\mathcal{F}$ have been replaced by calls to a single instance of ρ.

This is often used to examine the need for common setup parameters. If $\hat{\mathcal{F}}$ can be realized by a protocol in which many independent copies use use only one set of parameters, then we can implement all instances of $\mathcal{F}$ as given in that protocol with the same parameters and they will still operate as if they were completely independent.

We can also show that $\hat{\mathcal{F}}_{SOK}$ can be an explicit verification algorithm (which by Theorem 4.1 means we can use it to define the language for a higher level signature of knowledge functionality). Note that π^{Σ} as defined in Section 2.3 does not realize $\hat{\mathcal{F}}_{SOK}$. Suppose we send a signature query to an $\mathcal{F}_{SOK}$ instance for language L with ssid *ssid*. Then we send a verification query with the resulting signature to another $\mathcal{F}_{SOK}$ instance also for language L, but with slightly different ssid *ssid*$'$. If we are interacting with $\hat{\mathcal{F}}_{SOK}$, these two instances will be completely independent, so the second $\mathcal{F}_{SOK}$ instance (*ssid*$'$), by which this message has not been signed, will reject the signature. If we are interacting with π^{Σ}, however, any signature which has been generated for a language will always cause the verification procedure to accept when given the same language and the same CRS by the completeness property of SimExt-secure signatures. However, we can make a small modification to avoid this problem.

If we extend π^{Σ} so that instead of signing message m and verifying signatures on m, we sign $m \circ ssid$ and verify signatures on $m \circ ssid$, the resulting protocol will UC-realize $\hat{\mathcal{F}}_{SOK}$ for NP languages. Further, recall that π^{Σ} UC-realizes $\mathcal{F}_{SOK}(U)$. Thus, π^{Σ} uses the same verification algorithm for all languages. If we allow π^{Σ} to output this verification algorithm, then this protocol will UC realize the explicit verification $\hat{\mathcal{F}}_{SOK}$ functionality. We can similarly show that we can UC-realize the explicit verification $\hat{\mathcal{F}}_{SOK}(\mathcal{F}_0)$ functionality. Finally, this implies that we can realize the nested $\mathcal{F}_{SOK}$ functionality in which $\mathcal{F}_0 = \hat{\mathcal{F}}_{SOK}(\mathcal{F}_0')$.

In some cases, we would like to create a different type of multisession $\mathcal{F}_{SOK}$ functionality. In this multisession functionality, many instances of $\mathcal{F}_{SOK}$ may be operating simultaneously, however, they need not be completely independent. Instead, we allow some of the $\mathcal{F}_{SOK}$ instances to share the subfunctionalities they use to define their languages. Thus, we could have two $\mathcal{F}_{SOK}$ instances, SOK_1, and SOK_2, both issuing signatures of knoweldge on behalf of the language of messages signed by some party P. If we used the standard multisession functionality, each $\mathcal{F}_{SOK}$ instance would be defined in terms of a separate instance of $\mathcal{F}_{SIG}$. Thus, signatures which were generated and accepted by the $\mathcal{F}_{SIG}$ instance running within SOK_1 might be rejected by the $\mathcal{F}_{SIG}$ instance running withing SOK_2, even though they were all issued by party P. Thus, we need this alternative "shared subfunctionality" multisession extension. (Note that this becomes more useful when we allow languages defined by multiple subfunctionalities as described in Appendix 4.3.) We can show that any SimExtsecure signature of knowledge scheme can also be used to realize this type of

extension.

4.5 Delegatable Anonymous Credentials

Given any SimExt secure signature of knowledge scheme, i.e. any scheme that UC-realizes $\mathcal{F}_{SOK}(\mathcal{F}_0)$ and its shared-subfunctionality and multi-subfunctionality extensions (see Sections 4.3 and 4.4), we can build a delegatable anonymous credential system.

As an example, we will see how we can implement the scenario given in the introduction. We will show how to formulate anonymous credentials in terms of our multi-subfunctionality $\mathcal{F}_{SOK}$ functionalities. Since we have shown that any SimExt secure signature of knowledge scheme can be used to realize these functionalities, this will give an implementation of delegatable anonymous credentials based on any SimExt secure scheme.

At the base level of our delegation chain is the certification authority (CA), which has public key PK_{CA} and secret key SK_{CA}. The CA wishes to issue a credential to Brown employee with public key PK_{Emp} and secret key SK_{Emp}. In our delegatable anonymous credential scheme, this credential will be a traditional signature under the CA's public key on the message PK_{Emp}. We can desribe this in the UC model by saying that CA uses $\mathcal{F}'_{SIG}$(the explicit verification form of $\mathcal{F}_{SIG}$) with $sid = CA \circ sid'$ to issue credentials to Brown employees.

Now, we would like our Brown employee to be able to issue credentials to a guest with public key PK_{Guest} and secret key SK_{Guest}. In this case, we want the credential $Cred_{Guest}$ to be a signature on the guest's public key PK_{Guest} of knowledge of a valid employee credential for PK_{Emp} and a corresponding SK_{Emp}. In order to describe this signature of knowledge language in the UC model, we need two subfunctionalities. To verify the employee credential, we need a signature functionality as described above. To verify the public key, we extend the multisession encryption functionality $\mathcal{F}_{Enc}$ to an explicit verification algorithm by adding a (Verify x, w) query which returns true iff x is an encryption public key and w is the corresponding secret key, and a corresponding VerificationAlgorithm query.

We can now define the guest credential functionality as a shared-subfunctionality signature of knowledge functionality(see Section 4.3.) Let $\mathcal{F}_1 = \mathcal{F}'_{SIG}$ and $\mathcal{F}_2 = \mathcal{F}'_{Enc}$. When we issue a new credential, we want $w = (w_1, w_2, w_3) = (PK_{Emp}, SK_{Emp}, Cred_{Emp})$ i.e. all the information which must be hidden in the resulting signature. Since the credential should reveal no information besides the fact that it is valid, we'll let x (the potentially visible part of the signature) = "valid". Now we need only determine which inputs get passed to $\mathcal{F}_1$ and $\mathcal{F}_2$. Let $f_1(x, w = (w_1, w_2, w_3)) = w_1$ and $f'_1(x, w = (w_1, w_2, w_3)) = w_3$). Thus, the first step in checking whether (x, w) is in our language sends (`Verify` w_1, w_3), i.e. (`Verify` $PK_{Emp}, Cred_{Emp}$) to $\mathcal{F}'_{SIG}$, in other words, we check that $Cred_{Emp}$ is a valid signature on PK_{Emp}. Now let $f_2(x, w) = w_1$ and $f'_2(x, w) = w_2$. Then the second step sends (`Verify` w_1, w_2), i.e. (`Verify` PK_{Emp}, SK_{Emp}) to $\mathcal{F}'_{Enc}$, in other words, we check that SK_{Emp} is the secret key corresponding to public key PK_{Emp}. Finally, we define $b(x, w, \mathcal{F}_1(f_1(x, w), f'_1(x, w)), \mathcal{F}_2(f_2(x, w), f'_2(x, w))) = \mathcal{F}_1(f_1(x, w), f'_1(x, w)) \wedge \mathcal{F}_2(f_2(x, w), f'_2(x, w))$, so x, w is accepted by our language iff both functionalties accept. Call this Guest credential functionality $\mathcal{F}^1_{SOK}$. A Guest credential is now a signature of knowledge given by $\mathcal{F}^1_{SOK}$ on the message PK_{Guest}.

The taxi driver credentials are defined analagously. In this case, in the UC formulation, we represent the credentials as second level signatures of knowledge. Thus, credentials are now signatures of knowledge on behalf of a combination of $\mathcal{F}_1 = \mathcal{F}^1_{SOK}$ and $\mathcal{F}_2 = \mathcal{F}'_{Enc}$, $x =$"valid", and $w = (PK_{Guest}, SK_{Guest}, Cred_{Guest})$. f_1, f_2, and b are defined as above. A taxi credential is

represented by a signature by this new $\mathcal{F}_{SOK}$ functionality on message $m = PK_{Taxi}$.

5 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

[ACJT00] Giuseppe Ateniese, Jan Camenisch, Marc Joye, and Gene Tsudik. A practical and provably secure coalition-resistant group signature scheme. In Mihir Bellare, editor, *Advances in Cryptology — CRYPTO 2000*, volume 1880 of *Lecture Notes in Computer Science*, pages 255–270. Springer Verlag, 2000.

[BBS04] Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. In Matthew K. Franklin, editor, *Advances in Cryptology — CRYPTO 2004*, volume 3152 of *Lecture Notes in Computer Science*, pages 41–55. Springer Verlag, 2004.

[BdM94] Josh Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital signatures. In Tor Helleseth, editor, *Advances in Cryptology — EUROCRYPT '93*, volume 765 of *Lecture Notes in Computer Science*, pages 274–285. Springer-Verlag, 1994.

[BMW03] Mihir Bellare, Daniele Micciancio, and Bogdan Warinschi. Foundations of group signatures: Formal definitions, simplified requirements, and a construction based on general assumptions. In Eli Biham, editor, *Advances in Cryptology — EUROCRYPT 2003*, volume 2656 of *Lecture Notes in Computer Science*, pages 614–629. Springer Verlag, 2003.

[BR04] Mihir Bellare and Phillip Rogaway. The game-playing technique. `http://eprint.iacr.org/2004/331`, 2004.

[Can00] Ran Canetti. Security and composition of multi-party cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proc. 42nd IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 136–145, 2001.

[Can04] Ran Canetti. Universally composable signature, certification and authentication. `http://eprint.iacr.org/2003/239`, 2004.

[Can05] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocol. `http://eprint.iacr.org/2000/067`, 2005.

[Cha85] David Chaum. Security without identification: Transaction systems to make big brother obsolete. *Communications of the ACM*, 28(10):1030–1044, October 1985.

[CL01] Jan Camenisch and Anna Lysyanskaya. Efficient non-transferable anonymous multi-show credential system with optional anonymity revocation. In Birgit Pfitzmann, editor, *Advances in Cryptology — EUROCRYPT 2001*, volume 2045 of *Lecture Notes in Computer Science*, pages 93–118. Springer Verlag, 2001.

[CR03] Ran Canetti and Tal Rabin. Universal composition with joint state. In Dan Boneh, editor, *Advances in Cryptology — CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 265–281, 2003.

[Cra97] Ronald Cramer. *Modular Design of Secure yet Practical Cryptographic Protocol*. PhD thesis, University of Amsterdam, 1997.

[CS97] Jan Camenisch and Markus Stadler. Efficient group signature schemes for large groups. In Burt Kaliski, editor, *Advances in Cryptology — CRYPTO '97*, volume 1296 of *Lecture Notes in Computer Science*, pages 410–424. Springer Verlag, 1997.

[CvH91] David Chaum and Eugène van Heyst. Group signatures. In Donald W. Davies, editor, *Advances in Cryptology — EUROCRYPT '91*, volume 547 of *Lecture Notes in Computer Science*, pages 257–265. Springer-Verlag, 1991.

[DKNS04] Yevgeniy Dodis, Aggelos Kiayias, Antonio Nicolosi, and Victor Shoup. Anonymous identification in ad hoc groups. In Christian Cachin and Jan Camenisch, editors, *Advances in Cryptology — EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 609–626. Springer, 2004.

[DP92] Alfredo De Santis and Giuseppe Persiano. Zero-knowledge proofs of knowledge without interaction (extended abstract). In *33rd Annual Symposium on Foundations of Computer Science*, pages 427–436, Pittsburgh, Pennsylvania, 24–27 October 1992. IEEE.

[dSdCO+01] Alfredo de Santis, Giovanni di Crescenzo, Rafail Ostrovsky, Giuseppe Persiano, and Amit Sahai. Robust non-interactive zero knowledge. In Joe Kilian, editor, *Advances in Cryptology — CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 566–598. Springer Verlag, 2001.

[FLS99] Uriel Feige, Dror Lapidot, and Adi Shamir. Multiple noninteractive zero knowledge proofs under general assumptions. *SIAM Journal on Computing*, 29(1):1–28, 1999.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO '86*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer Verlag, 1987.

[GMR88] Shafi Goldwasser, Silvio Micali, and Ronald Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, April 1988.

[LRSW99] Anna Lysyanskaya, Ron Rivest, Amit Sahai, and Stefan Wolf. Pseudonym systems. In Howard Heys and Carlisle Adams, editors, *Selected Areas in Cryptography*, volume 1758 of *Lecture Notes in Computer Science*. Springer Verlag, 1999.

[Lys02] Anna Lysyanskaya. *Signature schemes and applications to cryptographic protocol design*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 2002.

[RST01] Ronald L. Rivest, Adi Shamir, and Yael Tauman. How to leak a secret. In Colin Boyd, editor, *Advances in Cryptology — ASIACRYPT 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 552–565. Springer Verlag, 2001.

[Sah99] Amit Sahai. Non-malleable non-interactive zero knowledge and adaptive chosen-ciphertext security. In *Proc. 40th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 543–553. IEEE Computer Society Press, 1999.

[SCP00] Alfredo De Santis, Giovanni Di Crescenzo, and Giuseppe Persiano. Necessary and sufficient assumptions for non-interactive zero-knowledge proofs of knowledge for all NP relations. In Ugo Montanari, José P. Rolim, and Emo Welzl, editors, *Proc. 27th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1853 of *Lecture Notes in Computer Science*, pages 451–462. Springer Verlag, 2000.

[Sho04] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. `http://eprint.iacr.org/2004/332`, 2004.

A UC definition

Here we will briefly review the UC framework as defined in [Can05].

In the UC model, each protocol is examined independently. Any other protocols operating simultaneously are represented by the environment. In the execution scenario, the parties participating in a protocol are given inputs by the environment, which captures the idea that input to a protocol can be dependent on all the other protocols running in the system. Each party's part in this protocol is represented by an ITM. These ITMs can communicate with each other as specified by the protocol. They can also invoke other ITMs to run subroutines. And finally, they can communicate with an adversary. This adversary gets input from and sends messages to the environment.

Security is defined by the description of an ideal functionality which specifies exactly what actions should be possible and what information leakage is admissible. Thus, this ideal functionality expects to receive input from parties, and deliver output back to some of them, while performing some sort of operation. The ideal functionality may also communicate with the adversary. It can send messages to the adversary, representing information that we allow a secure protocol to leak. It can also receive messages from the adversary, which represents information not under the control of the parties running the protocol.

We also define a simulator which will replace the adversary in the ideal world. This simulator may run a copy of the adversary, and its goal is to convert messages from the ideal world into the output that the environment expects from the adversary in the real protocol. This means that whatever information the environment expects to receive from the adversary is information that could be generated from what the ideal functionality leaks – the protocol leaks no more information than we have declared admissible.

We then define the ideal execution of the protocol, in which the environment sends input to the parties who forward it directly to the functionality. Any output from the functionality to the

parties is delivered to the environment. When the functionality tries to communicate with the adversary, the messages will be sent to and from the simulator instead.

Finally, a real protocol is considered secure if there exists a simulator such that for all environments and adversaries the environment cannot tell whether it is communicating with real parties running the real protocol ITMs and with the real adversary, or whether it is communicating with parties running the ideal protocol (forwarding all inputs to the ideal functionality) and with the simulator. In this case, we say that the real protocol UC-realizes the ideal functionality.

We now summarize a limited version of the UC-composition theorem. Suppose we are given a protocol ρ that UC-realizes an ideal functionality F. We are also given a protocol π which makes calls to the ideal protocol F, and which UC-realizes a functionality G. Let π^ρ be the functionality which runs π, but in which all calls to F are replaced by calls to ρ. Then the UC-composition theorem states that π^ρ UC-realizes G.

A few other details should be mentioned. Each ITM is identified by a unique identifier called an sid. The UC model allows ITMs to send messages to other ITMs running within the same protocol, or to subroutine ITMs. All messages must specify the sid of the ITM they are addressed to, and the code that it should run. In either case, if no ITM with the given sid exists, an ITM will be created running the given code. [3] (If the ITM is already running we ignore the given code.)

B Cannetti's Basic Signature Functionality

Over a series of papers, Canetti [Can00, Can01, Can04, Can05] gave several ideal functionalities for a signature scheme. His motivation was to capture the security properties that one would ideally want to obtain from a signature scheme. Several versions were proposed, each subsequent version an improvement over a previous one, giving the adversary less and less power. He then shows that a signature scheme realizes $\mathcal{F}_{SIG}$ if and only if it is existentially unforgeable against adaptive chosen message attack, as defined by Goldwasser, Micali, and Rivest [GMR88].

We refer to Canetti's most recent version [Can05], which differs from earlier versions in a few fundamental ways. The most important difference is the role that the ideal adversary now plays. In the 2000 formulation, the ideal adversary was contacted during signing, shown the message being signed and was asked to provide the formal signature σ. In the 2005 formulation, the adversary is contacted during key generation, and provides an algorithm Sign for generating signatures. The adversary is not contacted at all during signing, instead signatures are generated using Sign. This new variant captures the fact that signing happens inside the signer without interaction with the outside adversarial world, and so we need a notion of security that does not allow the adversary to know what messages were signed.

Another, more subtle but very important difference from the earliest versions [Can01], is that this functionality is allowed to produce an error message and halt, or quit, if things go wrong. That means it is trivially realizable by a protocol that always halts. As mentioned in 2.1, we only worry about protocols that realize our functionalities *non-trivially*, i.e. never output an error message.

[3] The code c includes the name of the functionality and the code for the real protocol which realizes it. If the new ITM is being invoked in the ideal world, the real code is ignored, and the new ITM runs the ideal functionality with the given name. If the new ITM is being invoked in the real world, the name is ignored and the new ITM runs the given code.

C Primitives

Recall that a cryptosystem $(\mathcal{G}, \mathsf{Enc}, \mathsf{Dec})$ is called *dense* if the following two distributions are statistically indistinguishable: (1) the uniform distribution on k-bit binary strings; (2) the distribution of public keys obtained by running $\mathcal{G}(1^k)$.

Recall that a non-interactive zero-knowledge (NIZK) proof system consists of algorithms ($\mathsf{NIZKProve}$, $\mathsf{NIZKSimsetup}, \mathsf{NIZKSim}, \mathsf{NIZKVerify}$).

$\mathsf{NIZKProve}$ takes as input (1) a common random string ρ; (2) the common input x; (3) the statement that is being proven about the common input x (i.e., the description of a poly-time non-deterministic Turing machine that accepts x); (4) the witness w that the statement is true. It outputs a proof π.

$\mathsf{NIZKSimsetup}$ generates a string ρ that is indistinguishable from random, together with some trapdoor information τ about this string. $\mathsf{NIZKSim}$ takes as input (1) the string ρ generated by $\mathsf{NIZKSimsetup}$; (2) the trapdoor τ; (3) the common input x; (4) the statement that is being proven about the common input x. It outputs a simulated proof π.

$\mathsf{NIZKVerify}$ takes as input (1) the common reference string ρ; (2) the proof π; (3) the common input x; (4) the statement about x that is being proven. It either accepts or rejects.

Such a proof system has has three basic properties: completeness ($\mathsf{NIZKVerify}$ always accepts proofs generated by $\mathsf{NIZKProve}$); soundness ($\mathsf{NIZKVerify}$ always rejects proofs of false statements); and zero-knowledge (a proof generated by $\mathsf{NIZKSim}$ is indistinguishable from one generated by $\mathsf{NIZKProve}$).

A multi-theorem NIZK proof system requires that these properties hold even as many proofs for adversarially (and adaptively) chosen statements are generated.

Here we need simulation-sound NIZK, which is a strengthening of the basic soundness property in the multi-theorem setting. Simulation-soundness requires that no probabilistic poly-time adversary can get $\mathsf{NIZKVerify}$ to accept a proof π for a false statement, even after obtaining *simulated* proofs (i.e. proofs produced by $\mathsf{NIZKSim}$ instead of $\mathsf{NIZKProve}$) of statements of its own choice.

We refer the reader to existing literature [FLS99, dSdCO+01] for formal definitions.