Digital Signatures from Probabilistically Checkable Proofs

by

Raymond M. Sidney

A.B., Harvard College, 1991

Submitted to the Department of Mathematics
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Mathematics

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1995

Author ....[illegible]........
Department of Mathematics
May 5, 1995

Certified by ..[illegible]........
Silvio Micali
Professor of Computer Science and Electrical Engineering
Thesis Supervisor

Certified by ....[illegible]........
Hartley Rogers, Jr.
Professor of Mathematics
Department Advisor

Accepted by ....[illegible]........
David A. Vogan
Professor of Mathematics
Chairman, Committee on Graduate Studies

Digital Signatures from Probabilistically Checkable Proofs

by

Raymond M. Sidney

Submitted to the Department of Mathematics
on May 5, 1995, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Mathematics

Abstract

We prove a very strong soundness result for CS proofs which enables us to use them as efficient, noninteractive proofs of knowledge for NP statements. We then apply CS proofs to digital signature schemes, obtaining a general method for modifying digital signature schemes so as to shorten the signatures they produce, for sufficently large security parameters.

Under reasonable complexity assumptions, applying our methods to factoring-based signature schemes yields schemes with $\theta(k^{1/2} \cdot \log^{-1/2} k)$ bits of security from $\theta(k)$-bit signatures; asymptotically, this compares favorably with the $\theta(k^{1/3} \cdot \log^{2/3} k)$ bits of security currently obtainable from traditional factoring-based $\theta(k)$-bit signatures.

Our technique can also be used to shorten the public keys needed to attain a given level of security.

Thesis Supervisor: Silvio Micali

Title: Professor of Computer Science and Electrical Engineering

Acknowledgments

I'd like to thank some of the people who helped to make me the man I am today.

From a purely physical perspective, I guess that would be my parents, my grandparents, my great-grandparents, and so on, plus any role models I've had in mind when I go to the weightroom.

More importantly, from a mental perspective, there're all the people who've had the (possibly undesired) opportunity to mold my intellectual so-called capabilities and my mental traits. Of course, my immediate forebears and other relatives all figure prominently here. Thanks for everything, Mom, Dad, Dan, Larry, and Jenny! You guys have supplied me with a lot of the motivation needed to get where I am today (wherever that is). My wife, Satomi Okazaki, has also been a big influence on me. She's been a great source of support and encouragement, and she's helped me a whole lot with my defense and other pleasant parts of grad school.

My advisor (well, "thesis supervisor," technically), Silvio Micali, has been a big source of information, discussion, and inspiration. He's the guy who really got me interested in theoretical cryptography in the first place. Thanks for everything, Silvio!

The other members of my thesis committee, Hartley Rogers, Jr. and Mauricio Karchmer, have not only helped me make this document more understandable than it would have been without them, but have also been willing to talk with me about various half-baked complexity-theoretic notions I've gotten during my years at MIT.

Other professorial types who I feel have had a lot of influence on my thoughts include (in roughly reverse temporal order) Dan Stroock, whose course on stochastic processes almost made a probabilist out of me; Tom Leighton, whose class on parallel algorithms gave me the idea of joining MIT's Theory of Computation group; Sy Friedman, who teaches a mean introductory logic class; Alexander Kechris, whose mathematical analysis class gave me some notion of what a rigorous proof should be; and Yaser Abu-Mostafa, who gave me a chocolate bar and taught me all about information theory.

Thanks also to Ron Rivest for the occasional little talk, particularly the one in which he suggested the contents of section 7.4.

Some non-professorial (at present) friends of mine that I'd like to thank are such luminaries as the Reverend Stanley F. Chen; Andrew "Chou-man" Chou; Andreas Coppi; my office-mate and colleague, Rosario Gennaro, who (among other things) has given me a lot of advice and feedback on my Meisterwerk; my office-mate, Marcos Kiwi, who has been very willing to share his knowledge in matters complexity-theoretic; David Moews; Bjorn Poonen; Alex Russell; Dan Spielman; Ravi "Koods" Sundaram; Marc Spraragen; and Ethan Wolf.

Finally, thanks to Phyllis Ruby for her patient assistance in dealing with MIT's considerable bureaucracy and helping me in many other ways. Thanks also to Anne Conklin, Bruce Dale, Be Hubbard, and David Jones.

I am extremely grateful to the Fannie and John Hertz Foundation for funding me throughout my graduate career with a very excellent Fellowship for Graduate Study. Without their support, I'd be a poor soul.

This thesis is dedicated in fond memory of Hu-bang.

Contents

Chapter 1

Introduction

Much of computational complexity theory is related to proof systems of various sorts. The class NP (languages recognizable in nondeterministic polynomial time) has long been characterized as the set of languages whose members contain short noninteractive proofs of membership. Relatively recently, it has been shown that PSPACE = IP (the set of languages recognizable in polynomial space coincides with the set of languages with short interactive proofs— see Lund, Fortnow, Karloff, and Nisan [20] and Shamir [27]) and NEXPTIME = MIP (the set of languages recognizable in nondeterministic exponential time coincides with the set of languages with short multiple-prover interactive proofs— see Babai, Fortnow, and Lund [5]).

Also recently, in [22], Micali introduced "CS proofs," a proof system which is in some ways more practical than previous proof systems. CS proofs are based on the *probabilistically checkable proofs* of Babai, Fortnow, Levin, and Szegedy [4] and Feige, Goldwasser, Lovász, Safra, and Szegedy [13] in a way which we shall describe later.

In all of these proof systems, we have the model of a prover (or several provers) trying to convince a verifier of some fact.

1.1 A cursory look at proofs of knowledge

At the same time that complexity theorists have been busy trying to use proof systems to prove that various complexity classes are equal or unequal, cryptographers and other researchers have been trying to put proof systems of various sorts to practical uses. Uses of zero-knowledge proof systems for NP languages in different protocols reveal that in addition to zero-knowledge proofs of membership, a second type of zero-knowledge proof exists: zero-knowledge proofs of knowledge. These are proof systems in which one party in a protocol wishes to "prove" to another party that it "knows" something (in particular, we imagine that the Prover wishes to prove to the Verifier that it knows a *short proof*, or *NP witness*, of some fact). See Tompa and Woll [30], Feige, Fiat, and Shamir [12], and De Santis and Persiano [9] for more information about zero-knowledge proofs of knowledge.

Now, it is more or less trivial to see exactly when and how normal NP proof systems for language membership can also be used as proofs of knowledge. And since probabilistically checkable proofs are really just NP witnesses which have been encoded for error-correction, the same statement holds for probabilistically checkable proof proof systems— a given probabilistically checkable proof essentially *contains* the NP witness that it proves knowledge of (we shall explain all this in much more detail later on).

However, for CS proofs of knowledge, as with zero-knowledge proofs of knowledge, the situation is more complicated. This is because these types of proofs do not explicitly contain the NP witness whose existence they purport to prove. Indeed, zero-knowledge proofs of knowledge should contain *no information* about what the NP witness is (in a technical sense); and CS proofs, as we shall see later, can be much too short to actually reveal much information about an NP witness.

1.2 Contributions of this thesis

In this thesis, we define what it means to use CS proofs as proofs of knowledge. We prove that a property which we call *strong computational soundness of CS proofs of knowledge* holds.

We are guided in doing this not only by results in the area of zero-knowledge proofs, but also by Micali's original intent in developing CS proofs. In particular, if we downsize CS proofs so that they only prove NP statements, we see that Micali wanted CS proofs to have the following two basic properties (among others):

1. It should be easy for the Prover to convince the Verifier of a true fact, if the Prover has an NP witness of that fact.

2. Unless the Prover has tremendous computational power, it should be very unlikely that it can convince the Verifier of any false fact.

As we see, these properties, while desirable, do not suffice for using CS proofs as proofs of knowledge. It could conceivably be easy to provide CS proofs of true statements, even if the Prover doesn't know "why" the statements are true.

Our proof of strong computational soundness of CS proofs of knowledge can be viewed as a kind of converse to property 1. Essentially, we show that:

> It is not much easier to find a CS proof of a statement than it is to find an NP witness of that statement.

In addition to giving meaning to CS proofs of knowledge, we give an application of CS proofs of knowledge. We show how to use them to make existing digital signature schemes[1] more efficient.

To do this, we present a general transformation which modifies digital signature schemes. In essence, we take a scheme in which one signs a message with some particular kind of signature string, and we change it into a scheme in which one signs a message by giving a *CS proof of knowledge* of that signature string. Thus, instead of supplying the original signature outright, the Signer supplies a different string—one which would be computationally difficult to find without knowledge of original signature.

As we shall see, the benefit of signing by providing a CS proof of knowledge of a signature, rather than by giving an actual signature, is that the proofs of knowledge can be much shorter than the strings that they prove knowledge of. Hence our approach can lead to digital signature schemes with smaller signature lengths.

[1]Digital signature schemes are a way of providing authentication for electronic messages. We shall discuss them at length in chapter 6.

Chapter 2

Preliminaries

In this chapter, we shall establish some notations, definitions, and conventions which we shall make use of. Once we have done this, we can enter into the technical matters at hand.

2.1 Notation

By "1^k" we mean the number k written in unary, that is, a string of k 1's.

If S, T, $\ldots$ are probabilistic algorithms, and $p(x, y, \ldots)$ is a predicate, then we denote by $\Pr(p(x, y, \ldots) : x \leftarrow S; y \leftarrow T; \ldots)$ the probability that $p(x, y, \ldots)$ holds after the execution of the assignments $x \leftarrow S$, $y \leftarrow T$, $\ldots$, respectively. Furthermore, if H is a finite set, we denote by $x \in_R H$ the act of setting x equal to a randomly chosen element of H.

If $v \in \{0, 1\}^*$ is a binary string, we denote by v_R the string obtained by reversing the order of the bits in v, and we denote by $|v|$ the length of v. In addition, if v is not the empty string, we define $\underline{v}$ to be the string obtained by performing a logical NOT operation on the last (least significant) bit of v. For example, $01011_R = 11010$, $|01011| = 5$, and $\underline{01011} = 01010$.

If A and B are events (i.e., subsets of a probability space), then A^c is the complement of A and $A \setminus B = A \cap B^c$.

$\mathbb{N} = \{0, 1, 2, \ldots\}$.

We shall make free use of $O(\cdot)$, $o(\cdot)$, and related notations. Recall that if D is an infinite subset of $\mathbb{N}$, and $f(\cdot)$ and $g(\cdot)$ are nonnegative real-valued functions such that $D = \text{domain}(f) \subseteq \text{domain}(g)$, then we write:

- $f = o(g)$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ (the limit is taken over $n \in D$).
- $f = O(g)$ if there exists a constant $c > 0$ such that for all sufficiently large $n \in D$, $f(n) \leq c \cdot g(n)$.
- $f = \Omega(g)$ if there exists a constant $c' > 0$ such that for all sufficiently large $n \in D$, $f(n) \geq c' \cdot g(n)$.
- $f = \omega(g)$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ (the limit is taken over $n \in D$).
- $f = \theta(g)$ if $f = O(g)$ and $f = \Omega(g)$.

We can also write statements like "$f(n) = g(\omega(h(n)))$," which means that there is some function $j(n)$ such that $j(n) = \omega(h(n))$ and $f(n) = g(j(n))$. In addition, we shall feel free to extend these asymptotic notations to multivariable functions whose domain $D \subseteq \mathbb{N}^a$ is such that for all $n \in \mathbb{N}$, D contains elements all of whose coordinates exceed n (if this condition on D doesn't hold, we cannot interpret the notion of the function's arguments *all* going to infinity).

If $\mathcal{K} \subseteq \mathbb{N}$, then any function $f(\cdot) : \mathcal{K} \mapsto \mathbb{N}$ such that $f(k)$ is computable in time polynomial in k and $f(k) = k^{O(1)}$ is called a *length function* for $\mathcal{K}$.

2.2 NP

Let $WL(\cdot)$ be a length function for $\mathbb{N}$, and let $P(\cdot,\cdot)$ be a deterministic polynomial time predicate (i.e., algorithm outputting either 0 or 1) such that $P(x,w) = 0$ whenever $|w| \neq WL(|x|)$. We define the language $L(P, WL)$ to be the following set of strings:

$$L(P, WL) = \{x \in \{0,1\}^* : \exists w \text{ such that } P(x,w) = 1\}.$$

We say that NP is the set of all such languages $L(P, WL)$. For a given $x \in L(P, WL)$, any w such that $P(x, w) = 1$ is called an *NP witness* that $(\exists w : P(x, w) = 1)$ (the name "$WL(\cdot)$" was chosen to stand for "witness length").

Intuitively, NP is the class of languages whose members all have proofs of membership which can be quickly verified by a deterministic Turing machine. Suppose we have a Prover and a Verifier, and we fix an NP language $L(P, WL)$. If $x \in L(P, WL)$, then the Prover can clearly convince the Verifier of this fact by sending it an NP witness that $(\exists w : P(x, w) = 1)$— upon receipt of any such w, the Verifier can quickly check that it is a genuine NP witness.

However, the simple protocol above actually convinces the Verifier of more than just the fact that $x \in L(P, WL)$ (i.e., the fact that $(\exists w : P(x, w) = 1)$). It also convinces the Verifier that *the Prover knows an NP witness that* $(\exists w : P(x, w) = 1)$. It may appear that this is a meaningless distinction (and it is, in some cases). But consider the following:

EXAMPLE. Let $h : \{0,1\}^* \mapsto \{0,1\}^*$ be a length-preserving, bijective mapping which can be computed in polynomial time, but which cannot be inverted quickly. Let $WL(n) = n$, and let $P(x, w) = 1$ iff $x = h(w)$. Then it is easy to see that $L(P, WL) = \{0,1\}^*$. So for any $x \in \{0,1\}^*$, the Prover need not actually do anything to convince the Verifier that $x \in L(P, WL)$; in a sense, there is nothing to prove. In contrast, it's *not* trivial for the Prover to convince the Verifier that it knows an NP witness that $(\exists w : P(x, w) = 1)$ (although it's not especially difficult, either; it suffices for the Prover simply to send the witness to the Verifier).

In general, let $L(P, WL)$ be any NP language, and take $x \in L(P, WL)$. Any proof of knowledge of an NP witness that $(\exists w : P(x, w) = 1)$ can also be considered to be a proof that $x \in L(P, WL)$; however, as our example shows, the converse does not always hold.

When we consider other types of "proof systems" for NP languages, we will also have to be careful about the distinction between proofs of membership and proofs of knowledge of NP witnesses, and things will get more complicated than they are with the simple "just send over an NP witness" protocol above.

2.3 Computing with circuits

Although we shall make some use of ordinary polynomial time deterministic and randomized algorithms, our basic model of computation is a probabilistic circuit: an ordinary Boolean circuit containing special nodes which allow it to make random coin tosses. In other words, a probabilistic circuit is a directed, acyclic graph with five kinds of nodes:

1. **Input nodes**, with indegree 0 and outdegree at least 1.

2. **Constant nodes**, with indegree 0 and outdegree at least 1.

3. **Gate nodes**, which can be any of:

    - 2-input, 1-output AND gates, with indegree 2 and outdegree at least 1.
    - 2-input, 1-output OR gates, with indegree 2 and outdegree at least 1.
    - 1-input, 1-output NOT gates, with indegree 1 and outdegree at least 1.

4. **Randomized nodes**, with indegree 0 and outdegree at least 1, and which output either 0 or 1 with probability $\frac{1}{2}$ each, independently from what any other nodes do.

5. **Output nodes**, with indegree 1 and outdegree 0.

The **size** of a circuit is the number of edges it has.

If we give labels to its inputs and outputs (so that we can distinguish them), a probabilistic circuit is readily seen to compute a *probabilistic function* from its inputs to its outputs. That is, if a probabilistic circuit has a input nodes and b output nodes, then each possible input value $x \in \{0,1\}^a$ determines a probability distribution on the set $\{0,1\}^b$ of possible output values in a natural way. For convenience, we shall generally omit the word "probabilistic," and refer to probabilistic circuits simply as "circuits."

The reason circuits are interesting is that they seem more or less to capture most notions of computing in a very concrete way. In particular, if an algorithm runs in t

steps, its computation can be simulated in a reasonably natural way by a circuit of size polynomial in t.

2.3.1 Special nodes

We shall also make use of more specialized circuits which possess other types of nodes (in addition to the five kinds we mentioned earlier). We view the inputs and outputs of a "special node" as being labelled, so that we can make a distinction between different input bits or different output bits of the node. If $f : \{0,1\}^a \mapsto \{0,1\}^b$ is any probabilistic function, a circuit could have nodes for evaluating $f(\cdot)$; such nodes have a distinct inputs, each with exactly one edge leading into it, and b distinct outputs, each with at least one edge leading out of it.

Special nodes can be much more general than this, however. A collection of several special nodes might exhibit a *linked* probabilistic behavior, whereby the probabilistic functions computed by the nodes are not independent. The most general possible behavior of a collection $v_1, v_2, \ldots, v_k$ of special nodes, where each v_i has a_i inputs and b_i outputs, is specified by a probability distribution on the Cartesian product $F = \prod_i F_i$, where

$$F_i = \{\text{All functions } f_i : \{0,1\}^{a_i} \mapsto \{0,1\}^{b_i}\}.$$

A circuit can also have *random oracle nodes.* These are more or less a specific case of what we just discussed; they are nodes computing some true random function mapping $\{0,1\}^a$ to $\{0,1\}^b$, for some a and b. They differ from a collection of b randomized nodes in that any two nodes for computing the same random oracle which take the same input must also have the same output. In other words, the nodes should be seen as providing oracle access to some particular randomly-chosen function. A circuit can have nodes for evaluating more than one random oracle.

There are two reasons that random oracles are not quite like other (possibly linked) probabilistic special nodes:

1. Any random oracle that a circuit has nodes for evaluating is *independent* (in

the sense of probability) of

- any other random oracles.
- the behavior of any randomized nodes.
- the behavior of any more exotic probabilistic special nodes.

2. *All* parties participating in a protocol have access to each random oracle.

Random oracles are therefore a source of common randomness in protocols. For a particular a-input, b-output random oracle, we envision all parties in a protocol as having access to a particular "black box"; when the protocol is begun, a specific function $f : \{0,1\}^a \mapsto \{0,1\}^b$ is chosen uniformly at random from all such functions, and is put into the box.

As we have indicated, *circuits* have nodes for evaluating such a random oracle; each such node will have a size "cost" of at least $(a+b)$ associated with it (since each input and each output has at least one edge attached to it). To allow an *algorithm* to have access to a random oracle, we give it a special "oracle tape" which it can write its oracle queries on. After writing a query, the algorithm then enters some special state; after one step, it leaves this state with the answer to its question written on the tape. The exact details of implementation are unimportant; all we really require is that a random oracle evaluation takes poly(a, b) steps.

2.3.2 Execution of circuits

Let C be a circuit with no input bits, containing any kinds of nodes. Make a list of all the special nodes in C, not including the random oracle nodes: we have nodes $v_1, v_2, \ldots, v_k$, where v_i has a_i inputs and b_i outputs. We recall that the behavior of all the v_i's can be specified by a probability distribution p on $F = \prod_i F_i$, where

$$F_i = \{\text{All [deterministic] functions } f_i : \{0,1\}^{a_i} \mapsto \{0,1\}^{b_i}\}.$$

Let's say that C has nodes for the k' random oracles $h_1 : \{0,1\}^{a_1} \mapsto \{0,1\}^{b_1}$, $h_2 : \{0,1\}^{a_2} \mapsto \{0,1\}^{b_2}$, ..., $h_{k'} : \{0,1\}^{a_{k'}} \mapsto \{0,1\}^{b_{k'}}$, and that C has Φ randomized nodes. Let (S_i, p_i) be the probability space of all functions $h_i : \{0,1\}^{a_i} \mapsto \{0,1\}^{b_i}$ under the uniform distribution, and let $(\$, p_\$)$ be the probability space of all Φ-bit strings under the uniform distribution.

Set Ω_C to be the product probability space

$$\Omega_C = (F, p) \times \prod_i (S_i, p_i) \times (\$, p_\$).$$

DEFINITION: **Global view**. An element of Ω_C is called a global view of C.

Let Ψ_C be the set of all functions mapping the edges of C to values in $\{0,1\}$. We see that there is a natural mapping from Ω_C to Ψ_C. This induces a probability measure on Ψ_C, enabling us to view Ψ_C as a probability space. In other words, if we choose particular instantiations for all random oracles in C, and we specify behaviors for all other nodes with probabilistic behavior, we determine everything "visible" that occurs in C.

DEFINITION: **Execution**. An element of Ψ_C is called an execution of C.

Note that an execution of C contains exactly the information we get if we look to see what goes into and out of every node in C. A global view of C contains all that and much more; for example, it contains every value of every random oracle in C (not just the values that C happens to evaluate).

If C is a circuit, then C can be executed by choosing an execution from the probability space Ψ_C. However, C can also be executed in a more on-line fashion. List the nodes of C in any order such that no node comes before any of its predecessors. Then we can evaluate the nodes of C one at a time, in this order, in a natural way. It's clear how to evaluate constant nodes, Boolean nodes, and output nodes. To evaluate a randomized node, output a random choice of either 0 or 1. To evaluate a random oracle node, check whether or not the same oracle query has already been asked. If so, output the same answer as was already returned; if not, output a randomly chosen binary string of the proper length. And finally, to evaluate any other special node, just choose its output according to the probability space (F, p), conditioned on what has been output previously.

The above procedure specifies precisely an execution of $\mathcal{C}$ (i.e., an element of $\Psi_{\mathcal{C}}$).

2.3.3 Subcircuits

It is sometimes useful to be able to consider part of a circuit as being a circuit in its own right.

DEFINITION: **Predecessor.** One node of a circuit is a predecessor of another if there is a directed path from the first node to the second.

DEFINITION: **Subcircuit.** If v is a node of $\mathcal{C}$ with nonzero indegree, and v is not an input node or a constant node, then let $\mathcal{D}$ be the subgraph of $\mathcal{C}$ induced by $\{v\} \cup \{\text{all predecessors of } v\}$. Obtain a new circuit $\mathcal{Z}$ by replacing v in $\mathcal{D}$ with a collection of output nodes, one node for each edge entering v. We call $\mathcal{Z}$ the subcircuit of $\mathcal{C}$ induced by v, and we write $\mathcal{Z} \preceq \mathcal{C}$. We adopt the convention that $\mathcal{C} \preceq \mathcal{C}$ as well, since $\mathcal{C}$ can (informally) be thought of as the subcircuit of $\mathcal{C}$ induced by $\mathcal{C}$'s outputs.

If we focus on some particular node v of $\mathcal{C}$ with nonzero indegree, and $\mathcal{Z}$ is the subcircuit of $\mathcal{C}$ induced by v, then we can view the execution of $\mathcal{C}$ (i.e., the choosing of an element from $\Psi_{\mathcal{C}}$) as occurring in three *stages* as follows:

1. $\mathcal{Z}$ is executed.

2. v's output is evaluated.

3. All remaining nodes' outputs are evaluated.

This is really just a slightly different way of viewing an on-line execution of $\mathcal{C}$.

2.3.4 Random variables and events

Set $\Sigma = \{0, 1, *\}$. We call the elements of Σ "components" in analogy with the way we call the elements of $\{0, 1\}$ "bits." For example, an element of Σ^k is a "k-component string."

A *random variable* on a probability space $\mathcal{S}$ (also called an $\mathcal{S}$-random variable) is a function mapping $\mathcal{S}$ to either $\mathbb{R}$ or Σ^*. If $\mathcal{Z} \preceq \mathcal{C}$, then any $\Psi_{\mathcal{Z}}$-random variable can also be viewed as an $\Omega_{\mathcal{Z}}$-random variable and a $\Psi_{\mathcal{C}}$-random variable; furthermore, any $\Omega_{\mathcal{Z}}$-random variable can also be viewed as an $\Omega_{\mathcal{C}}$-random variable.

We recall that an event in a probability space $\mathcal{S}$ is just a subset of $\mathcal{S}$. Events can also be thought of as being predicates on $\mathcal{S}$— we can say that an $\mathcal{S}$-event is an $\mathcal{S}$-random variable with range $\{0, 1\}$. Any event has some nonnegative probability of occurring. Since $\mathcal{S}$-events are $\mathcal{S}$-random variables, if $\mathcal{Z} \preceq \mathcal{C}$, then any $\Psi_{\mathcal{Z}}$-event can also be considered to be an $\Omega_{\mathcal{Z}}$-event and a $\Psi_{\mathcal{C}}$-event; furthermore, any $\Omega_{\mathcal{Z}}$-event can also be considered to be a $\Omega_{\mathcal{C}}$-event.

Chapter 3

Probabilistically checkable proofs

We now introduce a new kind of proof system which has been the subject of much important research in recent years. The setting for these proof systems is the same as for normal NP proofs of language membership (i.e., the Prover should send a single proof string which the Verifier can quickly check on its own); however, the proof string is encoded redundantly in such a way that the Verifier actually only needs to look at a little bit of it to judge whether or not it's a valid proof (Consult Spielman [28] to learn about the interesting connections between probabilistically checkable proofs and error-correcting codes).

Let us clarify this. If $L(P, WL)$ is an NP language, and $x \in L(P, WL)$, then the Verifier checks a probabilistically checkable proof π that $x \in L(P, WL)$ by performing the following procedure:

1. The **Verifier** flips some coins.

2. The Verifier uses its coin flips and the value x to determine some small number of bit positions in π to view.

3. The Verifier decides whether or not to accept π as a valid proof, based on everything it has seen so far (i.e., its coin flips; x; and some small number of bits of π).

3.1 Definitions and results

It will be well worth our while to be very precise in our discussion here of probabilistically checkable proofs. For our purposes, the definitions and results of Babai, Fortnow, Levin, and Szegedy [4] or Feige, Goldwasser, Lovász, Safra, and Szegedy [13] would essentially suffice; however, for the sake of efficiency, we follow the later work of Arora and Safra [3] and Arora, Lund, Motwani, Sudan, and Szegedy [2].

DEFINITION: $\mathbf{PCP}(\boldsymbol{r(n)}, \boldsymbol{q(n)})$. Let $r(n)$ and $q(n)$ be length functions for $\mathbb{N}$. The class PCP$(r(n), q(n))$ consists of all languages $L \subseteq \{0,1\}^*$ such that there exist a length function $R(n) = O(r(n))$ and a deterministic algorithm $T(\cdot,\cdot,\cdot)$ with random access to its third argument[1] such that:

1. $\forall x \in L$, $\exists \pi$ such that $\Pr(T(x,r,\pi) = 1 : r \in_R \{0,1\}^{R(|x|)}) = 1$.
2. $\forall x \notin L$ $\forall \pi$, $\Pr(T(x,r,\pi) = 1 : r \in_R \{0,1\}^{R(|x|)}) \leq \frac{1}{2}$.
3. $\forall x \in \{0,1\}^*$ $\forall r \in \{0,1\}^{R(|x|)}$, $T(x,r,\pi)$ examines only $O(q(|x|))$ bits of π. By this, we mean that the process of computing $T(x,r,\pi)$ could be performed in the following way:
   (a) Compute $T_1(x,r)$, which is a list $p_1, p_2, \ldots, p_Q$ of $O(q)$ bit positions in π.
   (b) Set $\hat{\pi} = \pi_{p_1} \circ \pi_{p_2} \circ \cdots \circ \pi_{p_Q}$ (doing this takes time $O(q)$, since the Verifier has random access to π). In other words, $\hat{\pi}$ contains the values of $O(q(|x|))$ specific bits of π.
   (c) Set $T(x,r,\pi) = T_2(x,r,\hat{\pi})$.

   Here, $T_1(\cdot,\cdot)$ and $T_2(\cdot,\cdot,\cdot)$ are both deterministic algorithms which run sufficiently quickly that $T(x,r,\pi)$ runs in time poly$(|x|, r(|x|), q(|x|))$. We make the natural convention that if any of the bit positions computed in step 1 is past the end of the string π, then $T(\cdot,\cdot,\cdot)$=0.

[1]We do not elaborate on this *random access* for the following reason: for our purposes, the third argument of $T(\cdot,\cdot,\cdot)$ will always have size polynomial in the size of the first argument; hence, random access to it can be accomplished in polynomial time anyway, even if only normal sequential access is supplied. In other words, our uses of the definition of $PCP(\cdot,\cdot)$ here would be unaffected by leaving out this "random access" qualifier.

If we have some $L \in \mathrm{PCP}(r(n), q(n))$, and a Prover is trying to convince a Verifier that some $x \in \{0,1\}^n$ is in L, then straightforward parsing of the definition of $\mathrm{PCP}(r(n), q(n))$ gives us a protocol PCP-PROOF$(\cdot)$ for proving that $x \in L$.

Protocol PCP-PROOF(x):

1. The Prover computes a "probabilistically checkable proof," π, that $x \in L$, and sends π to the Verifier.
2. The Verifier flips $R(n)$ coins to produce a random string r. Together, x and r determine $O(q(n))$ bit positions in π to examine.
3. The Verifier checks if $T(x, r, \pi)$ holds (looking only at those $O(q(n))$ bit positions in π). If so, the Verifier accepts the Prover's claim that $x \in L$; if not, the Verifier rejects the claim.

If $x \in L$, and the Prover sends an appropriate proof of this fact (which is guaranteed to exist), then the Verifier always accepts. This genre of property is usually referred to as a "completeness" property of the proof system under consideration. On the other hand, if $x \notin L$, then no matter what "proof" the Prover sends, the Verifier will reject it at least half of the time. This is called a "soundness" property of the proof system.

Observe that if we repeat steps 2 and 3 κ times each, then if $x \notin L$, the Verifier will reject with probability at least $(1 - 2^{-\kappa})$.

Of course, PCP-PROOF$(\cdot)$ requires the Verifier to receive the entire string π, which could have size superpolynomial in $|x|$. However, note that there are only $2^{R(|x|)}$ possible values for r, and for each one of them, $O(q(|x|))$ bits in π are examined. Hence we can assume that any probabilistically checkable proof π that $x \in L$ has length $O(2^{R(|x|)} \cdot q(|x|))$.

One of the major results shown by Arora, Lund, Motwani, Sudan, and Szegedy [2] is that $\mathrm{NP} = \mathrm{PCP}(\log n, 1)$. By the above remark, this means that if $L \in \mathrm{NP}$, then the statement $x \in L$ can be "proven" to a Verifier who flips only $O(\log |x|)$ coins and then examines only a *constant* number of bits in a proof of size polynomial in $|x|$ (indeed, it is shown by Polishchuk and Spielman [23] that a proof of size $O(|x|^{1+\epsilon})$ suffices). For a readable and well-annotated presentation of the $\mathrm{NP} = \mathrm{PCP}(\log n, 1)$

theorem and related work and applications, see Sudan [29].

3.2 PCPs as proofs of knowledge

We actually need a slightly stronger statement than is explicit in NP $=$ PCP$(\log n, 1)$. In particular, we are interested in using probabilistically checkable proofs as proofs of knowledge.

Let $WL(\cdot)$ be a length function, and let $P(\cdot,\cdot)$ be a deterministic polynomial time predicate such that $P(x,w) = 0$ whenever $|w| \neq WL(|x|)$. To use probabilistically checkable proofs as proofs of knowledge, it would suffice if, given any probabilistically checkable proof that $(\exists w : P(x,w) = 1)$, we could easily compute an NP witness of that fact. This is indeed the case, as is noted in Khanna, Motwani, Sudan, and Vazirani [18].

In the other direction, we note that from an NP witness that $(\exists w : P(x,w) = 1)$, we can quickly compute a probabilistically checkable proof of the same fact. So knowing an NP witness of some fact permits one to compute a probabilistically checkable proofof that fact, and conversely. We sum up everything we need to know about probabilistically checkable proofs in the following theorem, which is implicitly proved in the literature.

Theorem 1 ([3], [2]): *Let $WL(\cdot)$ and $P(\cdot,\cdot)$ be as above. Then there exist a constant $q \in \mathbb{N}$; a length function $R(n)$ such that $0 < R(n) = O(\log n)$; a length function $\pi L(\cdot)$ for $\mathbb{N}$; a polynomial time program $\pi(\cdot,\cdot)$ such that $|\pi(x,w)| = \pi L(|x|)$; a polynomial time program $W(\cdot,\cdot)$; and a deterministic program $T(\cdot,\cdot,\cdot)$ such that:*

- *$\forall x \in L(P, WL)$, if w is an* NP *witness that $\exists w : P(x,w) = 1$, then*

$$\Pr(T(x,r,\pi) = 1 : \pi \leftarrow \pi(x,w); r \in_R \{0,1\}^{R(|x|)}) = 1.$$

- *$\forall x\ \forall \pi$, if $\Pr(T(x,r,\pi) = 1 : r \in_R \{0,1\}^{R(|x|)}) \geq \frac{1}{2}$, then $W(x,\pi)$ is an NP witness that $(\exists w : P(x,w) = 1)$.*

- *$T(\cdot,\cdot,\cdot)$ runs in time polynomial in the length of its first input.*

- *$\forall x \in \{0,1\}^* \; \forall r \in \{0,1\}^{R(|x|)}$, $T(x,r,\pi)$ examines exactly q bits of π.*

Note that for a given $WL(\cdot)$ and $P(\cdot,\cdot)$, we have fixed the length of probabilistically checkable proofs that $(\exists w : P(x,w) = 1)$ to be $\pi L(x)$.

We shall use the notation from the above theorem (i.e., q, $R(\cdot)$, $T(\cdot,\cdot,\cdot)$, $W(\cdot,\cdot)$) in later chapters.

3.3 Credits

In this chapter, we have not presented any results of our own; instead, we have just explained the probabilistically checkable proof results in the literature which are needed to construct CS proofs of knowledge.

The characterization of NP as PCP$(\log n, 1)$ is the result of much research by many people in the areas of self-testing/correcting programs, interactive proof systems, and approximation of NP-hard functions. The papers most directly involved in proving this theorem include:

- Fortnow, Rompel, and Sipser [15], which first connects proof systems and oracle machines.

- Babai, Fortnow, and Lund [5], which proves that any NEXPTIME language has a multi-prover interactive proof system, thereby showing that (in modern terminology) NEXPTIME = PCP$(n^{O(1)}, n^{O(1)})$.

- Babai, Fortnow, Levin, and Szegedy [4], which "scales down" the results of [5] to produce easily-verified proofs of essentially any computation, demonstrating that NP $\subseteq$ PCP$(\log^{O(1)} n, \log^{O(1)} n)$.

- Feige, Goldwasser, Lovász, Safra, and Szegedy [13], which expands upon the results of [5] to demonstrate that NP $\subseteq$ PCP$(\log n \log\log n, \log n \log\log n)$.

- Arora and Safra [3], which introduces the notation "PCP$(\cdot,\cdot)$" and recursive proof-checking, and proves that NP = PCP$(\log n, \log^{O(1)} \log n)$.

- Arora, Lund, Motwani, Sudan, and Szegedy [2], which improves this further to NP $=$ PCP$(\log n, 1)$.

We stress that this is not meant to be a complete list; for much more historical information, see Sudan [29].

Chapter 4

Introducing CS Proofs

For our ultimate goal (reducing the lengths of digital signatures), we need to have a proof system which uses very short proofs. Let $WL(\cdot)$ and $P(\cdot, \cdot)$ be as usual. Then a probabilistically checkable proof that $(\exists w : P(x, w) = 1)$ generally has size at least as big as an NP witness w of this fact (i.e., size at least $WL(|x|)$).

With an eye towards shorter proofs (both of language membership and of knowledge of NP witnesses), we shall present the construction of Micali [22] for producing *CS proofs*, or *computationally-sound proofs*, out of probabilistically checkable proofs[1]. A CS proof of a statement is meant to be a noninteractive, easily verified, short proof. As we shall see, the brevity of CS proofs is not obtained without a price: it is possible for CS proofs of false statements to exist. However, CS proofs have what Micali calls a *computational soundness* property, meaning that finding a CS proof for a false statement, although possible, is computationally infeasible. We shall discuss different varieties of computational soundness in chapter 5; in the present chapter, we are more concerned with definitions and motivation than with theorems.

Our scenario is as follows. Fix some language $L(P, WL) \in$ NP. Let $x \in L(P, WL)$, and let w be an NP witness that $(\exists w : P(x, w) = 1)$. Set $n = |x|$. We wish to create a low-communication proof system for knowledge of a witness that $(\exists w : P(x, w) = 1)$, based on probabilistically checkable proofs.

[1]Actually, we modify the original construction a little bit to make slightly longer CS proofs; this will make it easier to prove our soundness theorem in section 5.3.

Actually, we should point out that CS proofs, like probabilistically checkable proofs, are useful for much more than just proving membership (or knowledge of witnesses) for NP languages; however, here, we present just the aspects of CS that we shall require for our purposes.

4.1 The basic idea behind CS proofs

The starting point for CS proofs is our simple protocol PCP-PROOF($\cdot$) on page 23. We want to take this protocol and modify it so that the Verifier doesn't need to perform as much communication.

4.1.1 Committing with mailboxes

Suppose that the Prover and the Verifier have a [physical] mailbox, for which the Verifier has the only key. Consider the protocol PCP-MAILBOX-PROOF($\cdot, \cdot$).

Protocol PCP-MAILBOX-PROOF(x, w):

1. The Prover computes $\pi = \pi(x, w)$.
2. For each of the $|\pi| = WL(n)$ bits π_i, the Prover writes on a separate piece of paper "Bit $\#i = \pi_i$." It then puts that piece of paper in the mailbox.
3. The Verifier flips $R(n)$ coins to determine $O(q(n))$ bit positions in π to examine. It asks the Prover for the values of those bits.
4. The Prover sends those bits to the Verifier.
5. The Verifier opens the mailbox, takes out the pieces of paper corresponding to proof bits that it wanted to see, and checks two things:
   (a) If any bit that the Prover sent it has a different value than the corresponding bit in the mailbox, the Verifier rejects the proof.
   (b) Otherwise, the Verifier decides whether to accept or reject based on the values of those bits, as in an ordinary probabilistically checkable proof.

In a sense, PCP-MAILBOX-PROOF($\cdot, \cdot$) and PCP-PROOF($\cdot$) barely differ (except that PCP-MAILBOX-PROOF($\cdot, \cdot$) is a protocol for performing proofs of *knowledge*).

However, if we don't consider all the effort the Verifier has to do in sifting through many pieces of paper to find specific pieces to be "communication," then the Verifier for PCP-MAILBOX-PROOF$(\cdot,\cdot)$ doesn't perform much communication at all.

Essentially, the Prover starts out by using the mailbox to *commit* to a probabilistically checkable proof; after this, there's no way for it to fool the Verifier later by pretending it had a different probabilistically checkable proof in mind. The Prover can *decommit* any part of the probabilistically checkable proof by sending its value to the Verifier; the Verifier then just needs to check that it agrees with what's "on file" in the mailbox.

4.1.2 Committing with random oracles

We need to have some way of simulating the committal above without the use of rather specialized postal hardware. As an alternative to a mailbox, let us have a security parameter[2] κ and a random oracle $f : \{0,1\}^{2\kappa} \mapsto \{0,1\}^{\kappa}$.

```
Function COMMIT$(1^\kappa, \pi)$:

  1. Append 0's to the end of $\pi$ until a string $\rho$ whose length is $\kappa$ times a power
     of 2 is obtained. Say $|\rho| = 2^\alpha \cdot \kappa$. Note that $|\rho| < 2|\pi|$, and so $\alpha = O(\log n)$.
  2. Divide $\rho$ into $2^\alpha$ segments of length $\kappa$ each, $\rho = \rho_0^0 \circ \rho_1^0 \circ \cdots \circ \rho_{2^\alpha-1}^0$ (we call
     the $\rho_i^0$'s the segments of $\pi$, or of $\rho$).
  3. Compress pairs of adjacent $\rho_j^0$'s together, meaning: compute $\rho_0^1 = f(\rho_0^0, \rho_1^0)$,
     $\rho_1^1 = f(\rho_2^0, \rho_3^0)$, ..., $\rho_{2^{\alpha-1}-1}^1 = f(\rho_{2^\alpha-2}^0, \rho_{2^\alpha-1}^0)$.
  4. Compress pairs of adjacent $\rho_j^1$'s together as above to compute $\rho_0^2 = f(\rho_0^1, \rho_1^1)$,
     $\rho_1^2 = f(\rho_2^1, \rho_3^1)$, ..., $\rho_{2^{\alpha-2}-1}^2 = f(\rho_{2^{\alpha-1}-2}^1, \rho_{2^{\alpha-1}-1}^1)$.
  5. Continue in this vein until an complete $2^\alpha$-leaf binary tree has been created,
     with $\mathcal{R} = \rho_0^\alpha$ at the root (see figure 4-1).
  6. Output the value $\mathcal{R} \in \{0,1\}^\kappa$ as a committal to the probabilistically check-
     able proof $\pi$.
```

[2]Many cryptographic protocols make use of ***security parameters***. Exactly what a security parameter means varies from protocol to protocol; in general, the larger the security parameter used is, the more resistant the protocol is to any kind of adversarial behavior. On the flip side, larger

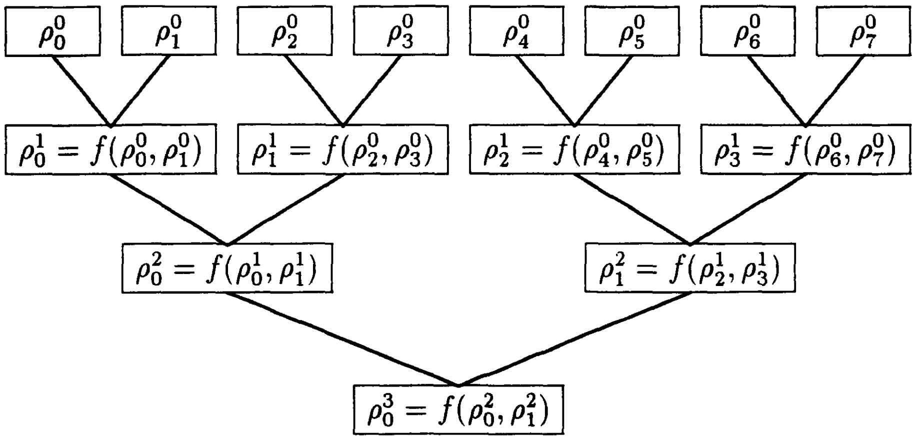
ρ_0^0
ρ_1^0
ρ_2^0
ρ_3^0
ρ_4^0
ρ_5^0
ρ_6^0
ρ_7^0
$\rho_0^1 = f(\rho_0^0, \rho_1^0)$
$\rho_1^1 = f(\rho_2^0, \rho_3^0)$
$\rho_2^1 = f(\rho_4^0, \rho_5^0)$
$\rho_3^1 = f(\rho_6^0, \rho_7^0)$
$\rho_0^2 = f(\rho_0^1, \rho_1^1)$
$\rho_1^2 = f(\rho_2^1, \rho_3^1)$
$\rho_0^3 = f(\rho_0^2, \rho_1^2)$

Figure 4-1: The tree computed to commit to π, if $\alpha = 3$

Suppose the Prover sends the Verifier a κ-bit string $\tilde{\mathcal{R}}$ as a committal. Later on, the Prover sends the Verifier a string π, claiming, "My value $\tilde{\mathcal{R}}$ was a committal to π." For COMMIT$(\cdot,\cdot)$ to be worth anything as a committal scheme, we would like it to be impossible for the Prover to decommit any string other than a string π which the Prover was actually thinking of, and on which it ran COMMIT$(\cdot,\cdot)$ to obtain its committal. This is not quite the case; however, it is *infeasible* for the Prover to pull off a crooked decommittal. Essentially, if the Prover can decommit to two distinct values relative to the committal $\mathcal{R}$, then it must have found a *collision* of $f(\cdot)$; to have probability $2^{-O(k)}$ of finding such a collision, the Prover must evaluate $2^{\Omega}(k)$ different values of $f(\cdot)$.

The type of commitment scheme we use here was first used in Merkle [21], for constructing a digital signature scheme. It was later used by Kilian in [19] for zero-knowledge proof protocols very similar to the "3-round pseudo-CS proofs" we present in section 4.2. Benaloh and de Mare [8] present an alternative method of committing to long strings which can be used in situations similar to ours.

security parameters also require more resources (computation, storage, communication, etc.) from the participants.

4.1.3 Decommitting part of a committed proof

It's not enough for us that COMMIT$(\cdot,\cdot)$ is a good way to produce a short committal to a string π. We need something more— a way to decommit *a single bit of* π without having to perform much communication. Let $0 \leq p < |\pi|$ be the index of a bit of π, and consider the function DECOMMIT$(\cdot,\cdot,\cdot)$ and the predicate CHECK-DCM$(\cdot,\cdot,\cdot,\cdot)$.

```
Function DECOMMIT$(1^\kappa, \pi, p)$:

  1. Compute the values $\rho_b^a$ which were computed in evaluating COMMIT$(1^\kappa, \pi)$.
  2. Set $i = \lfloor \frac{p}{\kappa} \rfloor$.
  3. Write the path (in the binary tree) from $\rho_i^0$ to the root $\rho_0^\alpha$ as $\rho_{j_0}^0$, $\rho_{j_1}^1$, ..., $\rho_{j_\alpha}^\alpha$, where $j_0 = i$ and $j_{u+1} = \lfloor j_u/2 \rfloor$.
  4. Set $D = \rho_{j_0}^0 \circ \rho_{\underline{j_0}}^0 \circ \rho_{j_1}^1 \circ \rho_{\underline{j_1}}^1 \circ \rho_{j_2}^2 \circ \rho_{\underline{j_2}}^2 \circ \cdots \circ \rho_{j_{\alpha-1}}^{\alpha-1} \circ \rho_{\underline{j_{\alpha-1}}}^{\alpha-1}$.
  5. Output $D$ as a decommittal of the bit $\pi_p$.
```

```
Predicate CHECK-DCM$(1^\kappa, \tilde{D}, p, \tilde{\mathcal{R}})$:

  1. Write $\tilde{D}$ as $\tilde{D} = A_0 \circ B_0 \circ A_1 \circ B_1 \circ \cdots \circ A_{\alpha-1} \circ B_{\alpha-1}$, where each $A_u$ and each $B_u$ has length $\kappa$.
  2. Check that for $u = 0, 1, \ldots, \alpha - 2$,
       $A_{u+1} = \begin{cases} f(A_u, B_u) & \text{if bit } \#u \text{ of } \lfloor p/\kappa \rfloor_R \text{ is a } 0 \\ f(B_u, A_u) & \text{if bit } \#u \text{ of } \lfloor p/\kappa \rfloor_R \text{ is a } 1 \end{cases}$.
  3. Check that $\tilde{\mathcal{R}} = \begin{cases} f(A_{\alpha-1}, B_{\alpha-1}) & \text{if bit } \#(\alpha-1) \text{ of } \lfloor p/\kappa \rfloor_R \text{ is a } 0 \\ f(B_{\alpha-1}, A_{\alpha-1}) & \text{if bit } \#(\alpha-1) \text{ of } \lfloor p/\kappa \rfloor_R \text{ is a } 1 \end{cases}$.
  4. If all $\alpha$ of the $f(\cdot)$ values computed check out as above, output 1 (accept the decommittal as valid); otherwise, output 0 (reject the decommittal).
```

The bottom line here is that it's about as difficult for a crooked Prover to convincingly decommit a bit that it wasn't thinking of when it made the committal $\mathcal{R}$ as it would be to decommit an entire string π that it wasn't thinking of. The basic reason for the difficulty is identical, too— the Prover would more or less have to find an f-collision. We won't go into more detail about this here, since chapter 5 will provide about all the detail one can handle.

We see that a single decommittal has length $2\alpha\kappa$.

4.2 3-round pseudo-CS proofs

We can take all the things we've built so far, and put them together in a 3-round proof system which needs very little communication, the protocol 3-ROUND PROOF$(\cdot,\cdot,\cdot)$, which is just a natural "conversion" of MAILBOX-PCP-PROOF$(\cdot,\cdot)$.

Protocol 3-ROUND-PROOF$(1^\kappa, x, w)$:

1. The Prover computes $\pi = \pi(x, w)$.
2. The Prover computes $\mathcal{R} = \text{COMMIT}(1^\kappa, \pi)$ and sends $\mathcal{R}$ to the Verifier.
3. The Verifier flips $R(n)$ coins, and uses the result to pick q bit positions $\pi_1, \pi_2, \ldots, \pi_q$ in the proof that it wishes to examine. It sends $\pi_1, \pi_2, \ldots, \pi_q$ to the Prover.
4. The Prover computes the decommittals $D_i = \text{DECOMMIT}(1^\kappa, \pi, p_i)$ for $i = 1, 2, \ldots, q$. It sends the Verifier $D_1 \circ D_2 \circ \ldots \circ D_q$.
5. The Verifier checks that CHECK-DCM$(1^\kappa, D_i, p_i, \mathcal{R}) = 1$ for $i = 1, 2, \ldots, q$. If any of these does not hold, the Verifier rejects the proof.
6. Otherwise, the Verifier decides whether to accept or reject based on the values of the bits decommitted, as in an ordinary probabilistically checkable proof.

Like PCP-PROOF$(\cdot)$, 3-ROUND-PROOF$(\cdot,\cdot,\cdot)$ can be repeated multiple times to obtain more security. For the sake of efficiency, the same committal $\mathcal{R}$ can be used for each repetition (i.e., only steps 3-6 need to be repeated). Furthermore, all the repetitions can be done in parallel, so that the entire protocol still takes only three rounds.

4.3 CS proofs, at last

It's now just a short step to CS proofs. In effect, CS proofs are obtained by taking the 3-ROUND-PROOF$(\cdot,\cdot,\cdot)$ protocol (repeated κ times), and replacing the Verifier's coin-flipping with a random oracle (to get rid of the interaction in 3-ROUND-PROOF$(\cdot,\cdot,\cdot)$).

We elaborate on this. Producing a CS proof with security parameter κ (a "κ-CS proof") that $(\exists w : P(x,w) = 1)$ requires two random oracles. One of them we have already seen, the random oracle $f : \{0,1\}^{2\kappa} \mapsto \{0,1\}^{\kappa}$. We now add in an additional random oracle, $g : \{0,1\}^{n} \times \{0,1\}^{\kappa} \mapsto \{0,1\}^{R(n)\cdot\kappa}$ (recall that $n = |x|$). Then the function CS-PROOF$(\cdot,\cdot,\cdot)$ can be used to create CS proofs.

```
Function CS-PROOF($1^\kappa, x, w$):

  1. Compute $\pi = \pi(x, w)$.
  2. Compute $\mathcal{R}$ = COMMIT($1^\kappa, \pi$).
  3. Evaluate $g(x \circ \mathcal{R})$ and divide it into $\kappa$ "challenges" of length $R(n)$ each,
     $g(x \circ \mathcal{R}) = r_1 \circ r_1 \circ \cdots \circ r_\kappa$.
  4. For each $s = 1, 2, \ldots, \kappa$,
     (a) Compute $p_1^s, p_2^s, \ldots, p_q^s$, the positions of the bits in $\pi$ that $T(\cdot,\cdot,\cdot)$ would
         wish to examine, given inputs $x$ and $r_s$.
     (b) Compute $D_i^s$ = DECOMMIT($1^\kappa, \pi, p_i^s$) for $i = 1, 2, \ldots, q$.
     (c) Set $h_s = D_1^s \circ D_2^s \circ \cdots \circ D_q^s$.
  5. Output the string $\mathcal{R} \circ r_1 \circ h_1 \circ r_2 \circ h_2 \circ \cdots \circ r_\kappa \circ h_\kappa$ as a $\kappa$-CS proof that
     $(\exists w : P(x, w) = 1)$.
```

Carefully going through the code for CS-PROOF$(\cdot,\cdot,\cdot)$ reveals one other difference from 3-ROUND-PROOF$(\cdot,\cdot,\cdot)$. Namely, CS proofs explicitly contain the random challenge bits, $g(x \circ \mathcal{R})$. In view of the fact that the string $\mathcal{R}$ is also contained in the CS proof, this might seem wasteful; however, it will simplify matters somewhat in section 5.3.

With everything we've put together so far, it's not hard to come up with a predicate CHECK-CS-PROOF$(\cdot,\cdot,\cdot)$ for the Verifier to use to check if a CS proof is valid.

```
Predicate CHECK-CS-PROOF($1^\kappa, x, M$):

1. Write $M$ as a concatenation $M = \tilde{\mathcal{R}} \circ \tilde{r}_1 \circ \tilde{h}_1 \circ \tilde{r}_2 \circ \tilde{h}_2 \circ \cdots \circ \tilde{r}_\kappa \circ \tilde{h}_\kappa$.
2. Divide $g(x \circ \tilde{\mathcal{R}})$ into $\kappa$ pieces of length $R(n)$ each, $g(x \circ \tilde{\mathcal{R}}) = r_1 \circ r_2 \circ \cdots \circ r_\kappa$.
3. For each $s = 1, 2, \ldots, \kappa$,
    (a) Check that $r_s = \tilde{r}_s$.
    (b) Write $h_s$ as a concatenation $h_s = D_1^s \circ D_2^s \circ \cdots \circ D_q^s$.
    (c) Compute $p_1^s, p_2^s, \ldots, p_q^s$, the positions of the bits in $\pi$ that $T(\cdot,\cdot,\cdot)$ would
        wish to examine, given inputs $x$ and $r_s$.
    (d) Check that CHECK-DCM($1^\kappa, D_i^s, p_i^s, \tilde{\mathcal{R}}$) $= 1$, for $i = 1, 2, \ldots, q$.
    (e) Check that the values decommitted for $\pi_{p_1^s}, \pi_{p_2^s}, \ldots, \pi_{p_q^s}$ would cause
        $T(\cdot,\cdot,\cdot)$ to accept the probabilistically checkable proof$\pi$, given inputs $x$
        and $r_s$.
4. If $M$ passes all the checks above, accept the CS proof $M$ (output a 1).
   Otherwise, reject $M$ by outputting a 0.
```

If CHECK-CS-PROOF$(x, M, \kappa) = 1$, we say that M is a *valid* κ-CS proof that $(\exists w : P(x, w) = 1)$. Note that this can hold even if there actually is no such witness (i.e., valid CS proofs are not necessarily correct).

A little bit of simple arithmetic reveals that the length of a κ-CS proof that $(\exists w : P(x, w) = 1)$ is precisely $\kappa[1 + R(n) + 2\alpha q\kappa] = O(\kappa^2 \log n)$. For convenience later on, we define the "CS Proof length" function $\Lambda_{P,W_L}(n, \kappa) = \kappa[1 + R(n) + 2\alpha q\kappa]$.

Chapter 5

CS proofs of knowledge

As with other proof systems, there are two distinct parts needed to demonstrate that CS proofs can actually serve as proofs:

1. Some kind of *completeness* property, which ensures that an honest prover can convince a Verifier of any particular true statement with very high probability.

2. Some kind of *soundness* property, which ensures that a dishonest prover can't convince a Verifier of any false statement with nontrivial probability.

As mentioned earlier, in [22], Micali is concerned with CS proofs as proofs of membership, and he demonstrates versions of these properties which he calls "feasible completeness" and "computational soundness." *Feasible completeness* means that not only can an honest Prover possessing an NP witness that $x \in L$ *always* successfully convince a Verifier that $x \in L$, but in addition, it can do so in time poly$(|x|, \kappa)$. *Computational soundness* means that a dishonest Prover, trying to convince a Verifier of some false statement $x \in L$, can only succeed with an exponentially small (in κ) probability, unless it has so much computational power that it can make exponentially many (in κ) oracle evaluations of the functions $f(\cdot)$ and $g(\cdot)$.

In this chapter, we shall present our results on how CS proofs can be used as proofs of knowledge. As always, this requires proving a *completeness* property and a *soundness* property.

Feasible completeness is sufficient for use in CS proofs of knowledge. We here present Micali's definition in a way which we feel clarifies how it can be used for to CS proofs of knowledge.

Let $WL(\cdot)$ and $P(\cdot,\cdot)$ be as usual (we fix $WL(\cdot)$ and $P(\cdot,\cdot)$ for the duration of this chapter). Then feasible completeness means that:

1. $\forall x \in \{0,1\}^*$, if w is an NP witness that $(\exists w : P(x,w) = 1)$, then
$$\Pr(\textsc{Check-CS-proof}(1^\kappa, x, M) = 1 : M = \text{CS-}\textsc{proof}(1^\kappa, x, w)) = 1.$$
2. The algorithms CS-$\textsc{proof}(\cdot,\cdot,\cdot)$ and $\textsc{Check-CS-proof}(\cdot,\cdot,\cdot)$ both run in polynomial time.

5.1 What about soundness?

Micali's computational soundness is not strong enough to show that CS proofs can be used as proofs of knowledge. We shall therefore propose and prove a new notion of soundness for CS proofs. In addition to being useful for its applications to digital signature schemes, as will be demonstrated in chapter 7, the notion is interesting in its own right.

If we consult the literature on zero-knowledge proofs, we see that there is a non-trivial difference between zero-knowledge proofs of membership and zero-knowledge proofs of knowledge (see Tompa and Woll [30], Feige, Fiat, and Shamir [12], and De Santis and Persiano [9]) which is similar to the distinction we face with CS proofs. It is not immediately obvious what it means for a protocol to "prove" that one of the parties "knows" an NP witness of some fact.

The three papers above all define a soundness property for zero-knowledge proofs of knowledge by saying that there must exist a "knowledge extractor," which is essentially an efficient algorithm which, given some form of control over running the Prover, can produce an NP witness of the fact to be proved.

The reason for defining soundness this way is that it's generally very difficult to formalize in a natural way what it means for an algorithm to "know" something. Given this, we might be tempted to make a "knowledge extractor" definition for CS proofs, as well. Since [9] is about *noninteractive* proofs of knowledge, and we find ourselves in a similar situation, it might seem reasonable to adapt their definition to our needs.

De Santis and Persiano present two types of "knowledge extractor" definitions for noninteractive zero-knowledge proofs of knowledge. The weaker definition involves the Prover having some particular $x \in \{0,1\}^*$ in mind, and trying to convince the Verifier that $(\exists w : P(x,w) = 1)$. The stronger definition allows the Prover to "shop around" and generate an $x \in \{0,1\}^*$ and a proof *together*, which could conceivably give the Prover more opportunity to falsely prove that it knows an NP witness of some fact. In analogy with their definitions, we shall make two similar definitions of soundness later in this chapter: "computational soundness of CS proofs of knowledge" and "*strong* computational soundness of CS proofs of knowledge."

However, the "knowledge extractor" approach is not precisely the best way to define soundness for CS proofs of knowledge. This is because, for CS proofs, as we shall see, performing committals via random oracle tree-hashing makes it possible to actually have some idea of what a Prover is "thinking." Given this, it is more intuitive to define soundness in terms of the Prover "having in mind" an actual NP witness. The resulting definitions of soundness, although slightly differently motivated, turn out to be essentially equivalent to the "knowledge extractor" definitions. In addition, Micali's original "computational soundness" property follows easily from either of our versions of soundness.

Most of the remainder of this chapter is devoted to formalizing and proving our new notion of soundness for CS proofs.

5.2 Setting the scene

Fix $n, b \geq 0$ and $\kappa \geq 1$. Let C be a circuit with no input bits and $(n + \Lambda_{P,W_L}(n, \kappa) + b)$ output bits. Suppose that C has s random-oracle nodes for $f : \{0,1\}^{2\kappa} \mapsto \{0,1\}^{\kappa}$, t

random-oracle nodes for $g : \{0,1\}^n \times \{0,1\}^\kappa \mapsto \{0,1\}^{R(n)\cdot\kappa}$, and any number of other special nodes of any types. Call $\mathcal{C}$'s $g(\cdot)$-nodes $G_1, G_2, \ldots, G_t$. Writing the output of $\mathcal{C}$ as $x \circ M \circ E$, where $|x| = n$, $|M| = \Lambda_{P,W_L}(n,\kappa)$, and $|E| = b$, we think of $\mathcal{C}$ as a circuit which tries to output a triple consisting of a value $x \in \{0,1\}^n$ (we call x an *instance*); a valid κ-CS proof that $(\exists w : P(x,w) = 1)$; and some b-bit auxiliary output, E.

DEFINITION: **f_i-node.** An f_i-node is an $f(\cdot)$-node which is a predecessor of G_i.

Let $\mathcal{C}_i$ be the subcircuit of $\mathcal{C}$ induced by G_i. Since G_i has $(n+\kappa)$ inputs, $\mathcal{C}_i$ is a circuit with no inputs and exactly $(n+\kappa)$ output nodes. We observe that two such subcircuits $\mathcal{C}_i$ and $\mathcal{C}_j$ need not be disjoint.

5.2.1 f-parents and implicit proofs

For the following definitions, let $\mathcal{Z} \preceq \mathcal{C}$, and let η be a κ-component $\Psi_{\mathcal{Z}}$-random variable (i.e., any function mapping $\Psi_{\mathcal{Z}}$ to Σ^κ).

DEFINITION: **f-left parent in $\mathcal{Z}$** and **f-right parent in $\mathcal{Z}$.** The f-left parent and the f-right parent of η in $\mathcal{Z}$ are defined as follows:

- If η is the output of at least one $f(\cdot)$-node in $\mathcal{Z}$, and if any two $f(\cdot)$-nodes in $\mathcal{Z}$ which output η both have the same input, then the f-left parent and the f-right parent of η in $\mathcal{Z}$ are the first and last κ bits of that common input, respectively.

- If η is the output of at least two $f(\cdot)$-nodes in $\mathcal{Z}$ with at least two distinct inputs, then the f-left parent and the f-right parent of η in $\mathcal{Z}$ are both equal to $*^\kappa$ (i.e., a sequence of κ $*$'s).

- If η is not the output of any $f(\cdot)$-node in $\mathcal{Z}$, then the f-left parent and the f-right parent of η in $\mathcal{Z}$ are both equal to $*^\kappa$.

The f-left parent of η and the f-right parent of η in $\mathcal{Z}$ are κ-component $\Psi_{\mathcal{Z}}$-random variables. For any $e \in \Psi_{\mathcal{Z}}$, if $\eta(e) = *^\kappa$, then the f-left parent and the f-right parent of η in $\mathcal{Z}$ also equal $*^\kappa$.

DEFINITION: **Implicit proof $\mathcal{Z}$ committed to with η.**

1. Set $\eta_0^\alpha = \eta$.

2. Define $\eta_0^{\alpha-1}$ and $\eta_1^{\alpha-1}$ to be the f-left and f-right parents of η_0^α in $\mathcal{Z}$, respectively.

3. Define $\eta_0^{\alpha-2}$ and $\eta_1^{\alpha-2}$ to be the f-left and f-right parents of $\eta_0^{\alpha-1}$ in $\mathcal{Z}$, and define $\eta_2^{\alpha-2}$ and $\eta_3^{\alpha-2}$ to be the f-left and f-right parents of $\eta_1^{\alpha-1}$ in $\mathcal{Z}$.

4. Continue in this fashion, defining η_{2b}^{a-1} and η_{2b+1}^{a-1} to be the f-left and f-right parents of η_b^a in $\mathcal{Z}$, until a tree similar to the one pictured in figure 4-1 is obtained, with η_b^a's instead of ρ_b^a's.

We say that the string $\tilde{\eta} = \eta_0^0 \circ \eta_1^0 \circ \cdots \circ \eta_{2^\alpha-1}^0$ is the implicit proof $\mathcal{Z}$ committed to with η.

$\tilde{\eta}$ is a $\Psi_{\mathcal{Z}}$-random variable of the same length that a probabilistically checkable proof that $(\exists w : P(x, w) = 1)$ should be (assuming that $|x| = n$), once that probabilistically checkable proof is padded to have length equal to κ times some power of 2. Indeed, as suggested by our terminology, the random variable $\tilde{\eta}$ is an attempt to reconstruct the probabilistically checkable proof that $\mathcal{Z}$ committed to via the short string η. Of course, for a particular random variable η, in a given execution $e \in \Psi_{\mathcal{Z}}$, it is quite possible that there is no such committed proof (i.e., that $\eta(e)$ was not computed by tree-hashing any kind of string), in which case $\tilde{\eta}(e)$ is likely to be just a string of $*$'s; more generally, there can be gaps in the committed proof, which are filled in with $*$'s in $\tilde{\eta}(e)$.

Note that a string of $*$'s in an implicit proof can mean either of two very different things:

1. It was impossible to "trace back" the committal string $\eta(e)$ to a consistent value for that part of the implicit proof, using only values of $f(\cdot)$ which $\mathcal{Z}$ had evaluated.

2. At some point in the process of trying to "trace back" from $\eta(e)$ to the implicit proof, there were at least two distinct possibilities of how to trace back.

The former event means that $\mathcal{Z}$ does not "know" some probabilistically checkable proof which tree-hashes to the committal $\eta(e)$. The latter event means that, later on, if $\eta(e)$ is used as a committal, then it might be feasible to decommit more than one value for a given proof bit. Because the latter event implies that $\mathcal{Z}$ has found an f-collision, it occurs very infrequently.

At long last, we define some particular random variables on $\Psi_{\mathcal{C}}$ and $\Psi_{\mathcal{C}_i}$:

- Write the output of $\mathcal{C}$ as $x \circ \tau \circ r_1 \circ h_1 \circ r_2 \circ h_2 \circ \cdots \circ r_\kappa \circ h_\kappa \circ E$, where $|x| = n$, $|\tau| = \kappa$, $|E| = b$, and for each $s = 1, 2, \ldots, \kappa$, $|r_s| = R(n)$ and $|h_s| = 2\alpha q\kappa$ (this breaks up $\mathcal{C}$'s output into an n-bit x; the pieces that a valid κ-CS proof that $(\exists w : P(x, w) = 1$ should contain; and an extra b-bit output). Recall that we already defined $M = \tau \circ r_1 \circ h_1 \circ r_2 \circ h_2 \circ \cdots \circ r_\kappa \circ h_\kappa$ to be the CS proof output by $\mathcal{C}$.

- For $i = 1, 2, \ldots, \kappa$:

  - Write the output of $\mathcal{C}_i$ (the input to G_i) as $x_i \circ \gamma_i$, where $|x_i| = n$ and $|\gamma_i| = \kappa$.

  - Let $\tilde{\gamma}_i$ be the implicit proof $\mathcal{C}_i$ committed to with γ_i.

Intuitively, we think of $\mathcal{C}_i$ as somehow computing an (instance, $\kappa -$ CS proof) pair (i.e., $\mathcal{C}_i$'s output), which will be used as input to the node G_i. Of course, the candidate κ-CS proof may not be a *valid* CS proof for the candidate instance. Nonetheless, each $g(\cdot)$-node does *a priori* give $\mathcal{C}$ a chance to output an instance and a valid κ-CS proof of knowledge for that instance.

x, M, E, τ, the r_s's, and the h_s's are $\Psi_{\mathcal{C}}$-random variables; and the γ_i's and the $\tilde{\gamma}_i$'s are $\Psi_{\mathcal{C}_i}$-random variables.

5.2.2 Anthropomorphization of circuits

For the duration of section 5.2.2, fix $\mathcal{Z} \preceq \mathcal{C}$.

Let β, β', β_0, and β_1 be κ-component $\Psi_{\mathcal{Z}}$-random variables. Let the $\Psi_{\mathcal{Z}}$-event "$\mathcal{Z}$ knows $f(\beta_0 \circ \beta_1) = \beta$)" be

$$\{e \in \Psi_{\mathcal{Z}} : \mathcal{Z} \text{ has an } f(\cdot)\text{-node which takes input } \beta_0(e) \circ \beta_1(e) \text{ and outputs } \beta(e)\}.$$

We similarly define the $\Psi_{\mathcal{Z}}$-events "$\mathcal{Z}$ knows the value of $f(\beta_0 \circ \beta_1)$" and "$\mathcal{Z}$ knows β' is an f-parent of β." Note that

$$\Pr(\beta = f(\beta_0 \circ \beta_1) | (\mathcal{Z} \text{ knows the value of } f(\beta_0 \circ \beta_1))^c) = 2^{-\kappa}$$

(the left-hand side of this equation is the conditional probability of an $\Omega_{\mathcal{Z}}$-event).

If λ and λ' are κ-component $\Psi_{\mathcal{Z}}$-random variables, we define a $\Psi_{\mathcal{Z}}$-event by saying that for any $e \in \Psi_{\mathcal{Z}}$, "$\mathcal{Z}$ knows that λ' is an ancestor of λ" holds if there is a sequence of κ-bit strings $\lambda'(e) = \lambda_0, \lambda_1, \ldots, \lambda_J = \lambda(e)$ such that $\mathcal{Z}$ knows that λ_j is an f-parent of λ_{j+1} for each $j = 0, 1, \ldots, J-1$.

Based on this event, we can define a $\Psi_{\mathcal{Z}}$-random variable, "the number of ancestors of λ that $\mathcal{Z}$ knows." Since any ancestor of λ that $\mathcal{Z}$ knows must be either the left half or the right half of the input to one of $\mathcal{Z}$'s $f(\cdot)$-nodes, and there are at most s such nodes, it follows that in any execution of $\mathcal{Z}$, $\mathcal{Z}$ cannot know more than $2s$ ancestors of λ.

Finally, if $\tilde{D}$ is a $2\alpha\kappa$-component $\Psi_{\mathcal{Z}}$-random variable, $0 \leq p < 2^{\alpha} \cdot k$, and $\tilde{\mathcal{R}}$ is a κ-bit $\Psi_{\mathcal{Z}}$-random variable, we define another $\Psi_{\mathcal{Z}}$-event by saying that for any $e \in \Psi_{\mathcal{Z}}$, "$\mathcal{Z}$ knows that $\tilde{D}(e)$ is a valid decommittal of bit #p relative to $\tilde{\mathcal{R}}(e)$" or equivalently, "$\mathcal{Z}$ knows $\text{CHECK-DCM}(1^{\kappa}, \tilde{D}(e), p, \tilde{\mathcal{R}}(e)) = 1$." This event is interpreted as meaning that two conditions hold:

1. $\mathcal{Z}$ knows all the values of $f(\cdot)$ that need to be evaluated to check the decommittal $\tilde{D}(e)$ relative to $\tilde{\mathcal{R}}$ (i.e., to run $\text{CHECK-DCM}(1^{\kappa}, \tilde{D}(e), p, \tilde{\mathcal{R}}(e))$) (recall that there are α such values).

2. These values of $f(\cdot)$ are such that if $\text{CHECK-DCM}(1^{\kappa}, \tilde{D}(e), p, \tilde{\mathcal{R}}(e))$ were actually run, it would output 1.

Note that

$$\Pr(\text{CHECK-DCM}(1^{\kappa}, \tilde{D}, p, \tilde{\mathcal{R}}) = 1 | (\mathcal{Z} \text{ knows CHECK-DCM}(1^{\kappa}, \tilde{D}, p, \tilde{\mathcal{R}}) = 1)^c) \leq 2^{-\kappa}$$

(the left-hand side of this equation is the conditional probability of an $\Omega_{\mathcal{Z}}$-event).

We can see this by taking $e \in \Psi_{\mathcal{Z}}$ and examining the two ways that $\mathcal{Z}$ can fail to know $\tilde{D}(e)$ is a valid decommittal:

1. $\mathcal{Z}$ knows all α of the values of $f(\cdot)$ that will be evaluated to check the decommittal $\tilde{D}(e)$ relative to $\tilde{\mathcal{R}}(e)$, and these values will not cause a Verifier to accept the decommittal as valid. In this case, the probability that $\tilde{D}(e)$ is a valid decommittal is 0.

2. $\mathcal{Z}$ does not know all α of the values of $f(\cdot)$ that will be evaluated to check the decommittal $\tilde{D}(e)$ relative to $\tilde{\mathcal{R}}(e)$. In this case, the probability that any value of $f(\cdot)$ that $\mathcal{Z}$ doesn't know will equal the "correct" value for it (i.e., the appropriate substring of $\tilde{D}(e)$, or $\tilde{R}(e)$ itself, as the case may be) is $2^{-\kappa}$.

5.2.3 Other events in $\Omega_{\mathcal{C}}$, $\Psi_{\mathcal{C}}$, and $\Psi_{\mathcal{C}_i}$

We define a few interesting events:

- VALID = $\{\text{CHECK-CS-PROOF}(1^\kappa, x, M)\}$. This is the event that $\mathcal{C}$ outputs an instance x and a valid κ-CS proof that $(\exists w : P(x, w) = 1)$.

- SAME$_i$ = $\{x = x_i\} \cap \{\tau = \gamma_i\}$. This is the event that $\mathcal{C}$'s output begins with the same instance and short committal that were given as input to G_i (i.e., the output of $\mathcal{C}_i$).

- DIFF = $(\bigcup_i \text{SAME}_i)^c$.

- COLL = {Some $f(\cdot)$-nodes in $\mathcal{C}$ have the same output, but distinct inputs}. This is the event that $\mathcal{C}$ finds an f-collision.

- COLL$_i$ = {Some f_i-nodes have the same output, but distinct inputs}. This is the event that $\mathcal{C}_i$ finds an f-collision. Note that COLL$_i \subseteq$ COLL.

- MORE$_i$ = {$\mathcal{C}$ knows more ancestors of γ_i than $\mathcal{C}_i$ does}. This is the event that $\mathcal{C}$ knows more useful values of $f(\cdot)$— useful in the sense of being potentially helpful in decommitting bits relative to γ_i— than $\mathcal{C}_i$ does.

VALID is an Ω_C-event, but the other events defined here are Ψ_C-events (which means they can also be viewed as Ω_C-events, of course). COLL$_i$ is actually a Ψ_{C_i}-event.

5.3 Strong computational soundness

For any $\tilde{\pi} \in \Sigma^{\tau L(n)}$ and $x \in \{0,1\}^n$, define

$$\Pi(x,\tilde{\pi}) = \Pr(T(x,r,\tilde{\pi}) = 1 : r \in_R \{0,1\}^{R(n)}).$$

We assume that if T examines any component of $\tilde{\pi}$ which is equal to $*$, then T rejects $\tilde{\pi}$ (i.e., outputs 0) outright. For convenience, we extend the notation $\Pi(\cdot,\cdot)$ to cover the case where $\tilde{\pi}$ has been extended to have length $2^\alpha \cdot \kappa$.

Let C be a circuit of the type we have been discussing. We now ask ourselves again what it means for C to know an NP witness that $(\exists w : P(x,w) = 1)$. Certainly it seems intuitive that C knows such a witness if the event

$$\text{KNOWS}_i = \text{VALID} \cap \text{SAME}_i \cap (\Pi(x_i,\tilde{\gamma}_i) \geq \tfrac{1}{2})$$

holds. After all, from an execution of C, one can easily compute x_i and $\tilde{\gamma}_i$; and if $(\Pi(x_i,\tilde{\gamma}_i) \geq \frac{1}{2})$, one can run $W(x,\tilde{\gamma}_i)$ to find an NP witness that $(\exists w : P(x,w) = 1)$.

There might be other, more subtle ways that C could know an NP witness that $(\exists w : P(x,w) = 1)$. However, as it turns out, this is the only way to know an NP witness that we need to consider. That is, let us define the event

$$\text{KNOWS} = (\bigcup_i \text{KNOWS}_i) \subseteq \text{VALID}.$$

Then we shall show in section 5.3.2 that the following property holds for CS proofs of knowledge:

DEFINITION: **Strong computational soundness of CS proofs of knowledge.** There exist positive constants c_1 and c_2 such that for sufficiently large κ, for any circuit C of the type we have been considering, if the total number of $f(\cdot)$-nodes and

$g(\cdot)$-nodes in C (i.e., $s+t$) is at most $2^{c_1 \cdot \kappa}$, then

$$\Pr(\text{VALID} \setminus \text{KNOWS}) \leq 2^{-c_2 \cdot \kappa}.$$

In effect, no circuit with fewer than exponentially many (in κ) random oracle nodes can output a valid pair (instance, κ-CS proof) without knowing a genuine NP witness (in the sense that the event KNOWS occurs), except with probability exponentially small (in κ).

Our strong computational soundness property clearly implies the following:

DEFINITION: **Computational soundness of CS proofs of knowledge.** This property is identical to strong computational soundness of CS proofs of knowledge, except that instead of quantifying over all circuits C with sufficiently few oracle nodes, we quantify over all such circuits C such that the instance x output by C is a *constant* (instead of permitting it to vary from execution to execution). That is, in effect, the value x is hard-wired into C.

5.3.1 The heuristics behind our proof

Before stating and proving our soundness results, we give in words the basic ideas behind them. Let C be any circuit of the usual type.

- It is very unlikely that C will find an f-collision (and hence, very unlikely that C_i will find an f-collision).

- As long as C_i doesn't find an f-collision, the value $x_i \circ \gamma_i$ is a committal to the value $\tilde{\gamma}_i$ as being a possible probabilistically checkable proof from which to create a κ-CS proof that $(\exists w : P(x_i, w) = 1)$.

- It is possible for C to output an instance x and a valid κ-CS proof M that $(\exists w : P(x, w) = 1)$, even if $x \circ M$ was never supplied as the input to any $g(\cdot)$-node. However, it is very unlikely.

- It is possible for C to output an instance x and a valid κ-CS proof M that $(\exists w : P(x, w) = 1)$ such that the committal τ of the CS proof is γ_i, but such that some of the decommittals in the CS proof are to segments which were not actually "committed to" (relative to γ_i) by $\mathcal{C}_i$. However, assuming that $\mathcal{C}_i$ did not find any f-collisions, this is also unlikely. There are two ways it can occur:

1. $\mathcal{C}$ can be lucky and know some ancestors of γ_i which $\mathcal{C}_i$ doesn't know, and which enable it to output a decommittal which it knows is valid.
2. $\mathcal{C}$ can be lucky and output a decommittal which it doesn't know is valid (but which *is* in fact valid).

5.3.2 The proof

Fix a circuit $\mathcal{C}$ of the usual type. Throughout section 5.3.2, probabilities are to be evaluated over the probability space $\Omega_{\mathcal{C}}$.

We intend to obtain our soundness results by progressing step by step, proving that the various ways of outputting an instance and a valid κ-CS proof of knowledge of an NP witness for it without "knowing" such an NP witness all have a very low probability of occurring.

Lemma 1 $\Pr(\text{VALID} \cap \text{DIFF}) \leq 2^{-\kappa}$.

Proof For $\mathcal{C}$'s output to be a valid κ-CS proof of *anything*, the strings $r_1 \circ r_2 \circ \cdots \circ r_\kappa$ and $g(x \circ \tau)$ must be identical. However, if we condition on the event DIFF occurring, then these strings are independently distributed, and moreover, $g(x \circ \tau)$ has a uniform distribution on strings of length $R(n) \cdot \kappa$. So

$$\Pr(\text{VALID} \cap \text{DIFF}) \leq \Pr(\text{VALID}|\text{DIFF}) = 2^{-(R(n)\cdot\kappa)} \leq 2^{-\kappa}.$$

■

Lemma 1 says that it's unlikely for $\mathcal{C}$ to output a pair (x, M), where M is a valid κ-CS proof of knowledge that $(\exists w : P(x, w) = 1)$, unless the x and the committal part of M were actually given as input to one of $\mathcal{C}$'s $g(\cdot)$-nodes.

Lemma 2 $\Pr(\text{COLL}_i) \leq \Pr(\text{COLL}) \leq s^2 \cdot 2^{-\kappa}$.

Proof Consider an on-line execution of $\mathcal{C}$ in which $\mathcal{C}$'s $f(\cdot)$-nodes are evaluated, one after the other, in such an order that each $f(\cdot)$-node is evaluated after all its predecessors. At the time when the j^{th} node is evaluated, $\mathcal{C}$ knows at most $j-1<s$ distinct values of $f(\cdot)$, and so the probability that evaluating the j^{th} node lets $\mathcal{C}$ know a collision of $f(\cdot)$ is less than $s\cdot 2^{-\kappa}$. Since there are s nodes, the probability of finding an $f(\cdot)$-collision is less than $s^2\cdot 2^{-\kappa}$. ∎

Lemma 3 *If A is any $\Psi_{\mathcal{C}_i}$-event, then* $\Pr(\mathrm{MORE}_i|A)\le s(2s+1)\cdot 2^{-\kappa}$.

Proof The proof of this is similar to the proof of lemma 2. To find a new ancestor of γ_i that $\mathcal{C}_i$ doesn't know, $\mathcal{C}$ must find a new f-parent either of γ_i itself, or of one of its ancestors known to $\mathcal{C}_i$. Since there are at most $2s$ such ancestors, and $\mathcal{C}$ has at most s $f(\cdot)$-nodes outside of $\mathcal{C}_i$ with which to find a new f-parent, our bound follows. ∎

Lemma 4 *If $X\in\{0,1\}^n$ and $\Gamma\in\Sigma^{2^{\alpha}\cdot\kappa}$, then*

$$\Pr(\mathrm{VALID}\cap\mathrm{SAME}_i|\mathrm{COLL}_i^c\cap(x_i=X,\tilde{\gamma}_i=\Gamma))\le s(2s+1)\cdot 2^{-\kappa}+[\Pi(X,\Gamma)+2^{-\kappa}]^{\kappa}.$$

Proof This is essentially a formalization of what was said in section 5.3.1. Given the event we are conditioning on in the above probability, for M to be a valid κ-CS proof of knowledge that $(\exists w: P(x,w)=1)$, one of two things must happen:

1. $\mathcal{C}$ must know more ancestors of Γ than $\mathcal{C}_i$ does— in other words, the event MORE_i must occur.

2. $\mathcal{C}$ must successfully answer the κ independent random challenges in $g(X\circ\Gamma)$. Each challenge can be answered successfully in two ways:

   (a) The challenge happens to be something that $\mathcal{C}$ knows how to answer; the probability of this is $\Pi(X,\Gamma)$.

   (b) The challenge is not something that $\mathcal{C}$ knows how to answer; in other words, at least one of the decommittals that $\mathcal{C}$ outputs is something that $\mathcal{C}$

does not know is valid. Such a decommittal is in fact valid with probability at most $2^{-\kappa}$, we recall.

■

Corollary 1 $\Pr(\mathrm{VALID} \cap \mathrm{SAME}_i | \mathrm{COLL}_i^c \cap (\Pi(x_i, \tilde{\gamma}_i) < \frac{1}{2})) \leq (2s^2 + s + 3) \cdot 2^{-\kappa}$.

Proof This follows from the fact that for $\kappa \in \mathbb{N}$, $(\frac{1}{2} + 2^{-\kappa})^\kappa \leq \frac{9}{4} \cdot 2^{-\kappa}$. ■

Lemma 5 *For any* $1 \leq i \leq t$,

$$\Pr(\mathrm{VALID} \cap \mathrm{SAME}_i \cap (\Pi(x_i, \tilde{\gamma}_i) < \tfrac{1}{2})) \leq (3s^2 + s + 3) \cdot 2^{-\kappa}.$$

Proof

$$\begin{aligned}
&\Pr(\mathrm{VALID} \cap \mathrm{SAME}_i \cap (\Pi(x_i, \tilde{\gamma}_i) < \tfrac{1}{2})) \\
&\quad \leq \Pr(\mathrm{VALID} \cap \mathrm{SAME}_i \cap (\Pi(x_i, \tilde{\gamma}_i) < \tfrac{1}{2}) \cap \mathrm{COLL}_i) \\
&\qquad + \Pr(\mathrm{VALID} \cap \mathrm{SAME}_i \cap (\Pi(x_i, \tilde{\gamma}_i) < \tfrac{1}{2}) \cap \mathrm{COLL}_i^c) \\
&\quad \leq \Pr(\mathrm{COLL}) + \Pr(\mathrm{VALID} \cap \mathrm{SAME}_i | (\Pi(x_i, \tilde{\gamma}_i) < \tfrac{1}{2}) \cap \mathrm{COLL}_i^c) \\
&\quad \leq s^2 \cdot 2^{-\kappa} + (2s^2 + s + 3) \cdot 2^{-\kappa}.
\end{aligned}$$

■

Lemma 5 bounds the probability that a valid (instance, κ-CS proof) pair is output for which the CS proof is not derived from a probabilistically checkable proof that C has "in mind."

Theorem 2 *CS proofs of knowledge enjoy strong computational soundness.*

Proof Let C be a circuit of the type we have been discussing. Then

$$\mathrm{VALID} = (\mathrm{VALID} \cap \mathrm{DIFF}) \cup \bigcup_i (\mathrm{VALID} \cap \mathrm{SAME}_i \cap (\Pi(x_i, \tilde{\gamma}_i) \geq \tfrac{1}{2}))$$

$$
\begin{aligned}
& \cup \bigcup_i (\text{VALID} \cap \text{SAME}_i \cap (\Pi(x_i, \tilde{\gamma}_i) < \tfrac{1}{2})) \\
= \; & (\text{VALID} \cap \text{DIFF}) \cup \text{KNOWS} \\
& \cup \bigcup_i (\text{VALID} \cap \text{SAME}_i \cap (\Pi(x_i, \tilde{\gamma}_i) < \tfrac{1}{2})).
\end{aligned}
$$

So

$$
\begin{aligned}
\Pr(\text{VALID} \setminus \text{KNOWS}) \; & \leq \; \Pr(\text{VALID} \cap \text{DIFF}) \\
& \quad + \Pr(\bigcup_i (\text{VALID} \cap \text{SAME}_i \cap (\Pi(x_i, \tilde{\gamma}_i) < \tfrac{1}{2}))) \\
& \leq \; 2^{-\kappa} + \sum_{i=1}^{t} (3s^2 + s + 3) \cdot 2^{-\kappa} \\
& = \; [t(3s^2 + s + 3) + 1] \cdot 2^{-\kappa}.
\end{aligned}
$$

■

5.4 Extracting NP witnesses

We now wish to use our proof of strong computational soundness to edge even closer to soundness definitions which are based on "knowledge extractors."

Let $\mathcal{C}$ be a circuit of the type we have been discussing. That is, $\mathcal{C}$ has s $f(\cdot)$-nodes, t $g(\cdot)$-nodes, no inputs, and outputs $x \circ M \circ E \in \{0,1\}^n \times \{0,1\}^{\Lambda_{P,WL}(n,\kappa)} \times \{0,1\}^b$. We define the $\Psi_{\mathcal{C}}$-random variable $w \in \{0,1\}^{WL(n)}$ as follows:

1. For $i = 1, 2, \ldots, t$, let $w_i = W(x, \tilde{\gamma}_i)$.

2. If there is a value i such that w_i is an NP witness that $(\exists w : P(x, w) = 1)$, then let $w = w_{i_0}$, where i_0 is the first such value.

3. If there is no such i, let $w = 1^{WL(n)}$.

Let WITNESS be the $\Psi_{\mathcal{C}}$-event $\{P(x, w) = 1\}$.

Lemma 6 $\Pr(\textit{VALID} \setminus \textit{WITNESS}) \leq [t(3s^2 + s + 3) + 1] \cdot 2^{-\kappa}$.

Proof From the definition of w, KNOWS $\subseteq$ WITNESS. Hence this follows from our proof of theorem 2. ■

Corollary 2 *Let A be any $\Omega_{\mathcal{C}}$-event. Then*

$$\Pr((A \cap \mathit{VALID}) \setminus (A \cap \mathit{WITNESS})) \leq [t(3s^2 + s + 3) + 1] \cdot 2^{-\kappa}.$$

Now, starting with $\mathcal{C}$, we can construct a circuit $\mathcal{C}'$ which does nothing more than execute $\mathcal{C}$ and then compute w. $\mathcal{C}'$ outputs $x \circ M \circ E \circ w$, and $\mathcal{C}'$ can be built "on top of" $\mathcal{C}$ by adding $\mathrm{poly}(s, t, n, \kappa)$ edges and ordinary deterministic Boolean nodes to $\mathcal{C}$ (this upper bound on how much new circuitry needs to be added is easy, but somewhat painful, to verify).

There are clear bijections between global views of $\mathcal{C}$ and global views of $\mathcal{C}'$, and between executions of $\mathcal{C}$ and executions of $\mathcal{C}'$. In particular, we can think of any event or random variable associated with $\mathcal{C}$ (i.e., any $\Psi_{\mathcal{C}}$- or $\Omega_{\mathcal{C}}$-event or -random variable) as also being associated with $\mathcal{C}'$, and conversely.

For the sake of our ultimate goal in chapter 7, we now need to put all of the above together, and also observe that the construction and results above possess a certain *uniformity*.

Corollary 3 *Let $\mathcal{C}(\cdot)$ be a circuit of the type which we have been considering, except that, instead of having no inputs, $\mathcal{C}(\cdot)$ takes an a-bit input, y. $\mathcal{C}(\cdot)$ can be viewed as being a collection of 2^a different circuits, $\{\mathcal{C}(y) : y \in \{0,1\}^a\}$, each of which takes no inputs. For each of these 2^a circuits, $\mathcal{C}(y)$, we can define $\Omega_{\mathcal{C}}(y)$ and events such as VALID(y), WITNESS(y), etc. Then we can construct a circuit $\mathcal{C}'(\cdot)$ "on top of $\mathcal{C}$," which inputs an a-bit value, y, and outputs $x \circ M \circ E \circ w$ as above; which has size no more than $\mathrm{poly}(s, t, n, \kappa)$ more than the size of $\mathcal{C}(\cdot)$; and such that for each $y \in \{0,1\}^a$, for each $\Omega_{\mathcal{C}}(y)$-event $A(y)$,*

$$\Pr((A(y) \cap \mathit{VALID}(y)) \setminus (A(y) \cap \mathit{WITNESS}(y))) \leq [t(3s^2 + s + 3) + 1] \cdot 2^{-\kappa}.$$

(The above probability can be thought of as being taken over $\Omega_C(y)$; over $\Omega'_C(y)$; over all global views of $C(a)$; or over all global views of $C'(a)$. All of these have the same meaning)

Fortunately for the reader, this is as far as we have to go with proofs of knowledge.

Chapter 6

Defining digital signature schemes

At this point, we can briefly steer away from proof systems. We wish to set the stage for our application of CS proofs of soundness, and so this chapter is concerned with formalizing exactly what digital signature schemes are, and what it means for them to be "secure."

6.1 The purpose of digital signature schemes

Suppose that two parties, Alice and Bob, are communicating in some fashion other than a face-to-face conversation. Whenever Bob receives a message m which was allegedly sent by Alice, he would like to have some method of assuring himself of the validity of the communication. Furthermore, Bob might want a method of holding Alice *accountable* for what she says: if Alice later denies that she sent Bob m, Bob wants to have some way of proving to a third party (such as a judge) that Alice really did send him m. In short, there are essentially three (not necessarily disjoint) kinds of security Bob might want to have:

1. *Sender authentication*: Bob can be certain that m really did originate with Alice.

2. *Message authentication*: Bob can verify that the message m is exactly what was sent by Alice.

3. *Accountability*: Bob can convince a third party of the above two facts— namely, that none other than Alice sent him precisely m.

Traditionally, if Alice sends Bob a handwritten note, her signature on the note is considered sufficiently difficult to forge that it is a "proof" (both to Bob and to a third party) that the note is really from her. It's somewhat debatable how well Alice's signature actually works for this purpose, and in any case, it's certainly questionable that it gives very trustworthy message authentication.

Digital signatures were first introduced by Diffie and Hellman in [10] as a digital analog to handwritten signatures. Digital signatures are intended to provide all of the types of security mentioned above (sender authentication, message authentication, and accountability) for *electronic* communications.

Precise procedures for producing and verifying signatures vary greatly among different digital signature schemes, but at a very high level, most schemes work essentially as follows:

1. Alice somehow obtains or generates a matching *public key* and *secret key*. Alice's secret key is something that only she knows, but her public key is required to be common knowledge.

2. To sign a string m, Alice uses m, her public key, and her secret key to compute a "signature" string.

3. To check a signature, Bob— or anyone else— runs some sort of *verification algorithm* on the message, the signature, and Alice's public key. The result of this computation tells him whether to accept or reject the signed message.

Because Alice is the only person who knows her secret key, only she can compute a valid-looking signature (hopefully!). solve the difficult problem arising from the message m and her public key. This is how a digital signature scheme gets its security, and is why digital signature schemes tend to be based on a family of difficult problems.

6.2 Digital signature schemes without security

As indicated, methods for producing and checking digital signatures tend to have certain important elements in common. We shall more or less follow the formalization of Goldwasser, Micali, and Rivest [17] in our definitions here. However, initially, our definition will be almost completely syntactic in nature— in particular, our definition of a digital signature scheme makes no mention whatsoever of security. We shall define the notion of security for a digital signature scheme soon afterward.

A digital signature scheme has a few components:

1. A *security parameter* is a number $k \in \mathbb{N}$. We permit the set $\mathcal{K}$ of legal values for k to be a proper subset of $\mathbb{N}$, as long as the set $\{1^k : k \in \mathcal{K}\}$ is recognizable in polynomial time. We shall often tacitly assume that $\mathcal{K}$ is an infinite set, so that we may consider asymptotic results about security.

2. For each $k \in \mathcal{K}$, the *message space* $\mathcal{M}_k$ is the set of all messages which can be signed for that particular value of k. Without any real loss of generality, we shall assume that $\mathcal{M}_k = \{0,1\}^{ML(k)}$ for all $k \in \mathcal{K}$, where $ML(k)$ is a length function for $\mathcal{K}$.

3. A *key generator* is a randomized algorithm $G(\cdot)$ which, on input 1^k for $k \in \mathcal{K}$, runs in expected time polynomial in k, and outputs a public key P_k of length $PL(k)$ and a secret key of length $SL(k)$, where $PL(k)$ and $SL(k)$ are length functions for $\mathcal{K}$.

4. A ***signing algorithm*** is a (possibly randomized) algorithm $\sigma(\cdot,\cdot,\cdot,\cdot)$ such that if P_k and S_k are matching public and private keys output by $G(1^k)$, and $m \in \mathcal{M}_k$, then $\sigma(1^k, P_k, S_k, m)$ runs in expected time polynomial in k, and outputs a signature $\sigma_m \in \{0,1\}^{\sigma L(k)}$ for m, where $\sigma L(k)$ is a length function for $\mathcal{K}$.

5. A *verifying algorithm* is a deterministic algorithm $V(\cdot,\cdot,\cdot,\cdot)$ such that if P_k is a public key output by $G(1^k)$, $m \in \mathcal{M}_k$, and $|\sigma| = \sigma L(k)$, then $V(1^k, P_k, m, \sigma)$ runs in time polynomial in k and outputs a 0 or 1 verdict on the validity of the

signature σ of the message m. 1 indicates a proper signature, and 0 indicates an improper one.

DEFINITION: **Digital signature scheme.** A digital signature scheme is a triple $(G(\cdot), \sigma(\cdot,\cdot,\cdot,\cdot), V(\cdot,\cdot,\cdot,\cdot))$, as above, such that if $k \in \mathcal{K}$ and $m \in \mathcal{M}_k$, then

$$\Pr(V(1^k, P_k, m, \sigma_m) = 1 : (P_k, S_k) \leftarrow G(1^k); \sigma_m \leftarrow \sigma(1^k, P_k, S_k, m)) = 1.$$

In other words, the essential part of a digital signature scheme is that a legitimate signature should always be recognized as being such. Recall that we have yet to say anything about security considerations; our definition as it stands does not preclude a rather useless digital signature scheme which produces the empty string as the signature for every $m \in \mathcal{M}_k$.

We also permit digital signature schemes to have access to random oracles. It may seem that a verifying algorithm with access to random oracles is no longer deterministic; however, we make a distinction between the "free randomness" of a randomized algorithm and the "common randomness" used by a deterministic algorithm with access to random oracles. We view an algorithm of the latter type as being deterministic, because anyone else with access to the same random oracles can predict exactly how the algorithm will behave. In contrast, a randomized algorithm's coin flips are private— unless it specifically sends them to someone else, no other parties will know their values.

Some important measures of a digital signature scheme's *efficiency* include the running times of these three programs (especially the last two, since a given Signer only needs to generate keys once), the lengths of the keys produced by $G(\cdot)$, the lengths of the signatures produced by $\sigma(\cdot,\cdot,\cdot,\cdot)$, and the sizes of the actual programs for $\sigma(\cdot,\cdot,\cdot,\cdot)$ and $V(\cdot,\cdot,\cdot,\cdot)$ (for applications where memory is at a premium, such as smart cards).

We shall often be somewhat informal in referring to digital signature schemes. For example, if S is a digital signature scheme, then S_k is "S with security parameter k," and an "S_k-signature of m" is what a Signer obtains by running $\sigma(1^k, P_k, S_k, m)$.

6.3 Security of digital signature schemes

We use the strongest natural model of security for digital signature schemes: security against an *existential forger* implementing an *adaptive chosen-message attack.* In simpler terms, a forger, Chet, starts out knowing only Alice's public key. He performs some computations on it, and comes up with a message m_1 whose signature he feels might be helpful to him. He gives m_1 to Alice, who gives him a (valid) signature σ_{m_1} for it. Chet then computes some more, and gets Alice to sign another message m_2 for him. He continues computing and gathering signatures in this fashion until, using all the information at his disposal, he finally computes a message m and a (hopefully valid) signature σ_m for it. Chet is successful in forging if his output (m, σ_m) is a valid signed message *which he never asked Alice to sign.*

Note that there are many other notions of security for digital signature schemes. [17] lists numerous other types of attacks. For example, Chet might not be permitted to obtain valid signatures from Alice, and he might have to be able to forge *any* message $m \in \mathcal{M}_k$, rather than just *some* single message; in that case, Chet would be a *universal forger* implementing a *key-only attack.*

For any reasonable notion of what constitutes successful forgery, we can make the informal definition that a digital signature scheme achieves b bits of security with security parameter k if no adversary with at most $2^{b/2}$ "units" of computational power can successfully forge with probability higher than $2^{-b/2}$. Now we shall formalize this definition.

6.4 Signing-oracle nodes

For the following definition, we assume that we have some particular digital signature scheme in mind.

DEFINITION: **Signing-oracle node**. A signing-oracle node for security parameter k is a special node which takes as input a message $m \in \mathcal{M}_k$ and outputs a signature $\sigma_m \in \{0,1\}^{\sigma L(k)}$ of m relative to a public key P_k, which is supplied as input to the circuit containing the node. In other words, a circuit containing a signing-oracle node should have exactly $PL(k)$ input bits, and the values of those bits is partially responsible for defining the node's computation. We describe in detail exactly how this computation works:

1. A secret key S_k which is compatible with P_k is chosen at random. That is, the probability distribution on the secret keys is the same as the probability distribution on secret keys output by $G(1^k)$, conditioned on $G(1^k)$'s public key output being P_k.

2. σ_m is then computed exactly as if $\sigma(1^k, P_k, S_k, m)$ were run.

3. If the circuit contains more than one signing-oracle node, then the same secret key S_k is used for all of them. That is, first, an S_k is chosen from the appropriate probability distribution. Then, for each signing-oracle node in the circuit, that same S_k, along with 1^k and P_k, is supplied to $\sigma(\cdot,\cdot,\cdot,\cdot)$, together with whatever input m the node has, to obtain a signature σ_m, which is the node's output.

6.5 Forging circuits and security levels

DEFINITION: **Forging circuit.** A forging circuit for security parameter k is a probabilistic circuit which, in addition to having the usual Boolean computation nodes (as well as nodes for any random oracles used by the digital signature scheme) can also have signing-oracle nodes. The forging circuit takes a public key, P_k, as input, does some computation with P_k (which can include obtaining signatures of messages relative to P_k— one signature per signing-oracle node), and outputs a forged message and signature $(m, \sigma_m) \in \{0,1\}^{ML(k)} \times \{0,1\}^{\sigma L(k)}$ (which may or may not be accepted as legitimate by $V(\cdot,\cdot,\cdot,\cdot)$). As indicated previously, the public key used by any oracle-signing nodes in the circuit is the same as the P_k passed as input to the circuit. We also call such a circuit a "k-forging circuit."

A k-forging circuit *successfully forges* if its output (m, σ_m) satisfies:

- $V(1^k, P_k, m, \sigma_m) = 1$.

- The circuit did not use m as the input to any of its signing-oracle nodes. In other words, the circuit did not explicitly receive a signature of m for free.

With these preliminaries in place, we are ready to define what security means in the context of digital signature schemes.

DEFINITION: **Security level.** Let $S = (G(\cdot), \sigma(\cdot,\cdot,\cdot,\cdot), V(\cdot,\cdot,\cdot,\cdot))$ be a digital signature scheme. We say that S achieves security level (N, p) for security parameter k if for any k-forging circuit $\mathcal{C}(\cdot)$ of size at most N,

$$\Pr(\mathcal{C}(P_k) \text{ successfully forges} : (P_k, S_k) \leftarrow G(1^k)) \leq p.$$

In other words, the probability of successful forgery with a circuit of size at most N is at most p. Naturally, this probability is taken over all random oracles, as well as over the values output by any randomized nodes and signing-oracle nodes in $\mathcal{C}(\cdot)$. For convenience, we say that a digital signature scheme *achieves b bits of security* for security parameter k if it achieves security level $(2^{b/2}, 2^{-b/2})$. We may also be lazy and say things like, "S_k achieves (N, p) security" or "S_k has b bits of security."

Observe how the above definition may be easily modified to obtain alternative notions of security. For example, if we did not permit forging circuits to contain signing-oracle nodes, then our definition would characterize security against existential forgers implementing a key-only attack— forgers who may not interact with the legitimate Signer to obtain valid signatures, and who must come up with a single valid forgery to be successful.

REMARK. At this point, many presentations of digital signature schemes might make an additional definition, calling a digital signature scheme *secure* if it achieves $\omega(k)$ bits of security for security parameter k. After all, if $\omega(k)$ bits of security are achieved, then the effort to forge grows superpolynomially with k, even though the efforts expended by the legitimate Signer and the Verifier remain polynomial in k. We do not make this definition, because we feel that it is extremely important to focus on the precise amount of effort a forger needs to expend. From a security viewpoint, there is a huge difference between a system which can be broken only with 2^k steps of computation, and a system which can be broken with $k^{\log\log k}$ steps of computation.

It is clear that any signature scheme which achieves security level (N, p) for a security parameter also achieves security level (N', p'), whenever $N' \leq N$ and $p' \geq p$. Similarly, any signature scheme which does *not* achieve security level (N, p) also does not achieve security level (N'', p''), whenever $N'' \geq N$ and $p'' \leq p$.

6.6 An example of a digital signature scheme

We give here a convenient example of a digital signature scheme. It is essentially a cross between Rabin's scheme (see [24]) and Williams's scheme (see [31]), with a random oracle thrown in so that it is possible to rigorously prove security results; both of the above schemes are variants of the RSA scheme presented by Rivest, Shamir, and Adleman in [26]. The security of all of these schemes is based on the presumed intractability of factoring a product of two large primes.

For our scheme, we will require a random oracle $f : \{0,1\}^{2k-2} \mapsto \{0,1\}^{2k-2}$. We also need to know a few simple number theoretic facts which have been observed in [31], [17] and elsewhere:

Let $\Pi_k = \{\text{All } k\text{-bit prime numbers}\}$. Define the subsets $F_k, G_k \subseteq \Pi_k$ by setting $F_k = \{p \in \Pi_k : p \equiv 3 \pmod 8\}$ and $G_k = \{q \in \Pi_k : q \equiv 7 \pmod 8\}$, and take $H_k = \{p \cdot q : p \in F_k \text{ and } q \in G_k\}$. If $n \in H_k$ and x is any residue modulo n (i.e., $x \in \mathcal{Z}_n^*$), then precisely one of the residues $\{x, -x, 2x, -2x\}$ is a square modulo n. Furthermore, any quadratic residue modulo n has exactly one square root which is itself a quadratic residue.

1. ***Legal security parameters.*** $\mathcal{K} = \{\text{All sufficiently large integers}\}$.

2. *Message space.* $\mathcal{M}_k = \{0,1\}^{2k-2}$.

3. *Key generation.* Upon input 1^k, $G(\cdot)$ chooses, uniformly at random, $p \in F_k$ and $q \in G_k$. The public key is the $2k$-bit product $n = p \cdot q \in H_k$, and the secret key is the concatenation $p \circ q$.

4. *Signatures.* We sign the message $m \in \{0,1\}^{2k-2}$ as follows:

(a) Let r be the [unique] element of $\{f(m), -f(m), 2f(m), -2f(m)\}$ which is a quadratic residue modulo n.

(b) Set σ_m to be the [unique] square root of r modulo n which is itself a quadratic residue.

5. *Verification.* To verify a signature σ_m of a message m relative to a public key P_k, we check that some element of $\{f(m), -f(m), 2f(m), -2f(m)\}$ is congruent to σ_m^2 modulo n.

For the above digital signature scheme, the capability of signing messages obviously requires the ability to compute square roots modulo n. It turns out that knowing the factorization of n enables one to do this quickly, and conversely— that is, if one can compute square roots modulo n quickly a reasonable fraction of the time, then one can factor n quickly.

The way the above scheme is set up, being able to compute square roots modulo n turns out to be more or less the only way to produce a signed message which will be accepted. It doesn't even do Chet any good to have Alice sign messages for him, unless Chet is fortunate enough to be able to use the same signature for two distinct messages. For example, if Chet can find $m_1 \neq m_2$ such that $f(m_1) = f(m_2)$, then he can ask Alice to sign m_1, and use the signature he obtains as a signature of m_2. However, given that $f(\cdot)$ is a random function, if Chet wants to be able to find such an m_1, m_2 with probability at least $2^{-2k/3}$, he will have to perform at least $2^{2k/3-1}$ evaluations of $f(\cdot)$.

Although we shall not do so here, it is not hard to formalize arguments like the above to show that complexity-theoretic assumptions about the difficulty of factoring imply that forging signatures in this scheme is difficult. A typical assumption to make about the difficulty of factoring is that for some particular $0 < c < 1$, for all sufficiently large k, for all probabilistic circuits $\mathcal{C}(\cdot)$ of size at most 2^{k^c} which have exactly $2k$ inputs and k outputs,

$$\Pr(\mathcal{C}(n) \in \{p, q\} : p, q \in_R \Pi_k; n = p \cdot q) \leq 2^{-k^c}.$$

Under such an intractability assumption, it is not difficult to prove rigorously that this scheme has $\Omega(k^c)$ bits of security.

Chapter 7

Derived digital signature schemes

Now we can put together everything we've seen so far. Looking at how we defined digital signature schemes, we realize that the valid signature for a message m produced by the Signer is more or less just an NP witness that $(\exists w : V(1^k, P_k, m, w) = 1)$. *Derived signature schemes* are obtained by having a signature be a *CS proofs of knowledge* of the above witness, rather than the witness itself.

7.1 Defining derived digital signature schemes

Let $S = (G(\cdot), \sigma(\cdot,\cdot,\cdot,\cdot), V(\cdot,\cdot,\cdot,\cdot))$ be a digital signature scheme such that the computation of $V(\cdot,\cdot,\cdot,\cdot)$ can be "factored" into a polynomial time oracle computation followed by a polynomial time oracle-free computation. By this, we mean that there exist algorithms $V_1(\cdot,\cdot,\cdot)$ and $V_2(\cdot,\cdot)$ such that:

- V_1 runs in deterministic polynomial time, and can make calls to any random oracles used by the signature scheme.
- V_2 also runs in deterministic polynomial time, but does not consult any random oracles.
- $V(1^k, P_k, m, \sigma) = V_2(V_1(1^k, P_k, m), \sigma)$.

Without loss of generality, we require that there be a *non-decreasing* length function for $\mathcal{K}$, $VL(\cdot)$, such that $|V_1(1^k, P_k, m)| = VL(k)$.

REMARK. It's not difficult to see that the sample digital signature scheme we presented in section 6.6 possesses this curious "factoring" property, as do many other schemes. In fact, the "factoring" property of $V(\cdot,\cdot,\cdot,\cdot)$ required above is trivially seen to hold for any deterministic polynomial time verifying algorithm which does not consult any random oracles.

Set

$$WL(n) = \left\{ \begin{array}{ll} \sigma L(k) & \text{if there is a } k \text{ such that VL(k)=n} \\ 0 & \text{otherwise} \end{array} \right\}$$

$$P(x,w) = \left\{ \begin{array}{ll} 1 & \text{if } |w| = WL(|x|) \text{ and } V_2(x,w) = 1 \\ 0 & \text{otherwise} \end{array} \right\}$$

We (slightly informally) define the *derived digital signature scheme*, S', to be a "2-parameter digital signature scheme" as follows:

1. *Legal security parameters.* A security parameter for S' is an ordered pair (k, κ), where $k \in \mathcal{K}$ and $\kappa \geq 1$.

2. *Message space.* $\mathcal{M}'_{(k,\kappa)} = \mathcal{M}_k$.

3. *Key generation.* $G'(1^k, 1^\kappa)$ has the same distribution as $G(1^k)$.

4. *Signatures.* Set $\hat{m} = V_1(1^k, P_k, m)$. Then $\sigma'(1^k, 1^\kappa, P_k, S_k, m)$ = a κ-CS proof that $(\exists \sigma : P(\hat{m}, \sigma) = 1)$ (with $\sigma_m = \sigma(1^k, P_k, S_k, m)$ as the NP witness).

5. *Verification.* $V'(1^k, 1^\kappa, P_k, m, \sigma') = 1$ if and only if σ' is a valid CS proof that $(\exists \sigma : P(\hat{m}, \sigma) = 1)$, where $\hat{m} = V_1(1^k, P_k, m)$.

Note that since $V_1(\cdot,\cdot,\cdot)$ is deterministic, there is a unique $\hat{m}$ which both the Signer and the Verifier can compute, and so they can each perform the computations they're supposed to be able to do. In fact, $G'(1^k, 1^\kappa)$, $\sigma'(1^k, 1^\kappa, P_k, S_k, m)$, and $V'(1^k, 1^\kappa, P_k, m, \sigma')$ all run in time poly(k, κ).

Of course, technically, S' is not a proper signature scheme (according to our definition). Nonetheless, we shall effectively consider it to be one, and shall freely talk about things such as the security it achieves for security parameter (k, κ).

7.2 Security of derived signature schemes

We now investigate the security of these variants of digital signature schemes. In particular, we would like to determine some relationship between the security of a digital signature scheme, and the security of the corresponding derived scheme.

Suppose we have a signature scheme S and a corresponding derived scheme S'. Say there is a forging circuit $\mathcal{F}'(\cdot)$ (with access to the random oracles $f(\cdot)$ and $g(\cdot)$, as well as to any random oracles that S_k uses) of size N', such that

$$\Pr(\mathcal{F}'(P_k) \text{ successfully forges an } S'_{(k,\kappa)}\text{-signature} : (P_k, S_k) \leftarrow G'(1^k, 1^\kappa)) = p'.$$

What we shall do is construct a circuit $\mathcal{F}(\cdot)$, which has size "not too much bigger than N'," and which forges S_k-signatures "about as well as" $\mathcal{F}'(\cdot)$ forges $S'_{(k,\kappa)}$-signatures. This will imply that forging S' signatures isn't much easier than forging S-signatures.

Set s and t equal to the number of $f(\cdot)$-nodes and $g(\cdot)$-nodes, respectively, in $\mathcal{F}'(\cdot)$. In addition,let μ be the number of signing oracle nodes in $\mathcal{F}'(\cdot)$.

Our construction of $\mathcal{F}(\cdot)$ shall take place in several steps, during which we construct intermediate circuits $\mathcal{F}_1(\cdot), \mathcal{F}_2(\cdot), \ldots$. To help keep things straight, the outputs and random variables of circuit $\mathcal{F}_i(\cdot)$ shall have the subscript "i," as shall events pertaining to $\mathcal{F}_i(\cdot)$.

STEP 1. Let $\mathcal{F}_1(\cdot) = \mathcal{F}'(\cdot)$.

Admittedly, this step isn't a very big step forward; it's just so we can take stock of our situation. In particular, $\mathcal{F}_1(\cdot)$ takes a $PL(k)$-bit input, P_k, and produces an $(ML(k) + \Lambda_{P,WL}(VL(k), \kappa))$-bit output, $m_1 \circ \sigma_1{}'$.

For a given P_k, define the following events on $\Omega_{\mathcal{F}_1(P_k)}$:

$$\begin{aligned} \text{VALID}_1(P_k) &= \{\text{CHECK-CS-PROOF}(\kappa, \hat{m}_1, \sigma_1{}') = 1\}, \text{ where } \hat{m}_1 = V_1(1^k, P_k, m_1). \\ \text{NEW}_1(P_k) &= \{m_1 \text{ was not the input to any of } \mathcal{F}_1(P_k)\text{'s signing-oracle nodes}\}. \end{aligned}$$

Note that $(\text{VALID}_1(P_k) \cap \text{NEW}_1(P_k))$ is the event that $\mathcal{F}_1(P_k)$ successfully forges.

STEP 2. Take each signing-oracle node in $\mathcal{F}_1(\cdot)$ and replace it with:

1. A signing-oracle node with the same inputs, and which computes S_k-signatures (as opposed to $S'_{(k,\kappa)}$-signatures).

2. Circuitry to compute a $S'_{(k,\kappa)}$-signature from the S_k-signature output by the S_k-signing-oracle node.

Call the resulting circuit $\mathcal{F}_2(\cdot)$.

Set s' and t' equal to the number of $f(\cdot)$-nodes and $g(\cdot)$-nodes, respectively, in $\mathcal{F}_2(\cdot)$. Note that $s' \neq s$ and $t' \neq t$, unless $\mu = 0$, since we need to use random oracle nodes to compute $S'_{(k,\kappa)}$-signatures. In any case, however, s' is no more than $\text{poly}(\mu, k, \kappa)$ bigger than s, and t' is no more than $\text{poly}(\mu, k, \kappa)$ bigger than t.

STEP 3. Add to $\mathcal{F}_2(\cdot)$ circuitry to compute $\hat{m}_2$ from m_2. In addition, reorder the outputs of $\mathcal{F}_2(\cdot)$ to produce a circuit $\mathcal{F}_3(\cdot)$, which inputs P_k and outputs $\hat{m}_3 \circ \sigma_3{}' \circ m_3$, where $\hat{m}_3 = V_1(1^k, P_k, m_3)$.

Regard $\mathcal{F}_3(\cdot)$ as being a circuit $\mathcal{C}(\cdot)$ of the type discussed in corollary 3. That is, $\mathcal{F}_3(\cdot)$ is a collection of $2^{PL(k)}$ circuits, each taking no input, and each of which, when executed, outputs a candidate (instance, κ-CS proof) pair, $\hat{m}_3 \circ \sigma_3{}'$, together with an "extra" output, m_3.

We can define the events NEW3(P_k), VALID3(P_k), and WITNESS3(P_k) on $\Omega_{\mathcal{F}_3(P_k)}$ in a natural way, exactly as in section 5.4.

STEP 4. Apply corollary 3 to $\mathcal{C}(\cdot) = \mathcal{F}_3(\cdot)$, obtaining a circuit $\mathcal{C}'(\cdot)$ (which we shall call $\mathcal{F}_4(\cdot)$, instead).

$\mathcal{F}_4(\cdot)$ takes an input P_k, and outputs $\hat{m}_4 \circ \sigma_4{}' \circ m_4 \circ \sigma_4$

Notice that, for a given P_k, there is *not* an obvious isomorphism mapping global views of $\mathcal{F}_1(P_k)$ to global views of $\mathcal{F}_4(P_k)$, or mapping executions of $\mathcal{F}_1(P_k)$ to executions of $\mathcal{F}_4(P_k)$. Nonetheless, it's clear that all of the circuits $\mathcal{F}_i(\cdot)$ are actually very closely related. The events VALID4(P_k), NEW4(P_k), and WITNESS4(P_k), defined on the space of global views of $\mathcal{F}_4(P_k)$, have extremely close parallels with corresponding events defined for previous circuits $\mathcal{F}_i(P_k)$.

Now, by corollary 3, $\Pr(\text{NEW}_4(P_k) \cap \Pr(\text{WITNESS}_4(P_k))$ is at least

$$\Pr(\text{NEW}_4(P_k) \cap \text{VALID}_4(P_k)) - [t'(3s'^2 + s' + 3) + 1] \cdot 2^{-\kappa}.$$

Saying that $\mathcal{F}'(\cdot)$ forges successfully with probability exceeding p' is equivalent to saying that

$$\sum_{P_k} \Pr(P_k) \cdot \Pr(\text{NEW}_1(P_k) \cap \text{VALID}_1(P_k)) > p',$$

where $\Pr(P_k)$ is the probability that $G'(\cdot,\cdot)$, on input (k, κ), outputs P_k as a public key. This statement, in turn, is equivalent to a similar statement about $\mathcal{F}_3(\cdot)$:

$$\sum_{P_k} \Pr(P_k) \cdot \Pr(\text{NEW}_3(P_k) \cap \text{VALID}_3(P_k)) > p'.$$

Everything comes together when we take two more steps:

STEP 5. Modify $\mathcal{F}_4(\cdot)$ so that it no longer outputs $\hat{m}$ and σ'. Call the result $\mathcal{F}_5(\cdot)$.

$\mathcal{F}_5(\cdot)$ can be seen to be a forging circuit for S_k, except that it contains random oracle nodes for evaluating $f(\cdot)$ and $g(\cdot)$. Furthermore, $\mathcal{F}_5(\cdot)$ successfully forges with probability exceeding

$$p' - [t'(3s'^2 + s' + 3) + 1] \cdot 2^{-\kappa}.$$

STEP 6. Replace the $f(\cdot)$-nodes and $g(\cdot)$-nodes in $\mathcal{F}_5(\cdot)$ with *simulated* oracle nodes to construct a circuit $\mathcal{F}_6(\cdot)$. In other words, each simulated random oracle node should just output a random value, except that whenever two simulated random oracle nodes receive the same input, they should have the same output.

At long last, we have our final circuit, $\mathcal{F}(\cdot) = \mathcal{F}_6(\cdot)$. The size of $\mathcal{F}(\cdot)$ is no more than $\text{poly}(s, t, \mu, k, \kappa)$ more than N'; and the probability that $\mathcal{F}(\cdot)$ successfully forges [an S_k signature] is less than p' by at most $\text{poly}(s, t, \mu, k, \kappa) \cdot 2^{-\kappa}$ (the polynomial bound on the size of $\mathcal{F}(\cdot)$ can be easily, but painfully, verified).

Theorem 3 *If $S'_{(k,\kappa)}$ does not achieve (N', p') security, then S_k does not achieve $(\text{poly}(N', k), p' - \text{poly}(N', k) \cdot 2^{-\kappa})$ security.*

Proof This follows from the fact that s, t, μ, and κ are all less than N'. ■

It's a simple matter now to restrict our derived signature schemes to being ordinary, 1-parameter schemes.

Corollary 4 *Let S be a digital signature scheme, with associated derived digital signature scheme S'. Let $b(k) = \omega(\log k)$ be such that for all $k \in \mathcal{K}$, S_k achieves $b(k)$ bits of security; and let $b'(k) \geq b(k)$. Define the digital signature scheme $T_k = S_{(k,b'(k))}$. Then T_k achieves $\Omega(b(k))$ bits of security.*

Proof By theorem 3, we can find a constant c such that for sufficiently large N' and k, if the scheme $S'_{(k,\kappa)}$ does not achieve (N', p') security, then S_k does not achieve $((N'k)^c, p' - (N'k)^c \cdot 2^{-\kappa})$ security. Setting $N = (N'k)^c$ and $p = p' - (N'k)^c \cdot 2^{-\kappa}$, and taking the contrapositive of this, we obtain:

> If $N^{1/c} \cdot k^{-1}$ and k are large enough, and if S_k achieves (N, p) security, then $S'_{(k,\kappa)}$ achieves $(N^{1/c} \cdot k^{-1}, p + N \cdot 2^{-\kappa})$ security.

We know that S_k achieves $(2^{\omega(\log k)}, 2^{\omega(\log k)})$ security. Any $2^{\omega(\log k)}$ function grows faster than k does; hence we can apply our contrapositive theorem. We find that for large enough k, $S_{(k,\kappa)}$ achieves $(2^{b(k)/2c} \cdot k^{-1}, 2^{-b(k)/2} + 2^{b(k)/2} \cdot 2^{-\kappa})$ security.

Since $\kappa \geq b(k)$, we see that for large enough k, T_k achieves $(2^{b(k)/2c} \cdot k^{-1}, 2 \cdot 2^{-b(k)/2})$ security. Given that $b(k) = \omega(\log k)$, this implies that T_k achieves $\Omega(b(k))$ bits of security. ■

7.3 Signature length for signature schemes

Recalling that a κ-CS proof that $x \in L(P, WL)$ has length $O(\kappa^2 \log |x|)$, we see that a T_k-signature of a message m has length $O(b'(k)^2 \log k)$. That is, our 1-parameter derived digital signature scheme produces signatures of length $O(b'(k)^2 \log k)$ to achieve $\theta(b(k))$ bits of security.

Consider once more our sample digital signature scheme from section 6.6, which we shall call S. For a given security parameter k, all permissible messages have length

$\theta(k)$; public keys, secret keys, and signatures have length $\theta(k)$ as well. How many bits of security do these signatures provide? At present, the best available factoring algorithms can factor k-bit numbers in $\exp(O(k^{1/3} \cdot \log^{2/3} k))$ steps (see Adleman's survey paper [1] for more information). Thus S_k has $O(k^{1/3} \cdot \log^{2/3} k)$ bits of security for $\theta(k)$ bit signatures. If we set $b'(k) = \theta(k^{1/3} \cdot \log^{2/3} k)$, we see that the scheme T_k has the same bit security (up to a constant factor) as S_k, but produces signatures of length $O((k^{1/3} \cdot \log^{2/3} k)^2 \cdot \log k) = O(k^{2/3} \cdot \log^{7/3} k)$.

Of course, the easier factoring turns out to be, the more it pays to use CS proofs as signatures, and the more savings can be gained in signature lengths. The "break-even" point occurs if $\theta(k)$-bit signatures supply $\theta(k^{1/2} \cdot \log^{-1/2} k)$ bits of security; however, as mentioned in the previous paragraph, it is *already* known that factoring is easier than that.

We can examine the security/signature length relationship in the other direction, as well— what security do S_k and T_k have for signatures of a given size $\theta(k)$? If we assume that S_k actually does achieve $\theta(k^{1/3} \cdot \log^{2/3} k)$ bits of security, then we can see that T_k achieves $\theta(k^{1/2} \cdot \log^{-1/2} k)$ bits of security.

7.4 Public key length for signature schemes

As suggested by Rivest [25], in addition to decreasing signature length, CS proofs can also be used to decrease the size of user's public keys. We show here what we mean.

Let S be a digital signature scheme such that $V(\cdot, \cdot, \cdot, \cdot)$ can be "factored," in the sense that there exist algorithms $V_1(\cdot, \cdot)$ and $V_2(\cdot, \cdot, \cdot)$ such that:

- V_1 runs in deterministic polynomial time, and can make calls to random oracles.
- V_2 runs in deterministic polynomial time, and does not consult any random oracles.
- $V(1^k, P_k, m, \sigma) = V_2(V_1(1^k, m), P_k, \sigma)$.

This is a slightly stronger condition on $V(\cdot, \cdot, \cdot, \cdot)$ then we encountered for constructing derived schemes, because it does not permit any oracle calls to depend on the value

P_k. However, once again, both our sample digital signature scheme and any trapdoor scheme with a deterministic, oracle-free verifying algorithm satisfy the conditions here.

Let $b(k) = \omega(\log n)$ be such that S_k achieves $b(k)$ bits of security, and let $b'(k)$ satisfy $b'(k) \geq b(k)$.

Let $H(\cdot)$ be a polynomial time computable collision-free hash function which, for each $k \in \mathcal{K}$, maps $PL(k)$-bit inputs to $b'(k)$-bit outputs (the intent is to use $H(\cdot)$ to hash public keys to obtain shorter public keys). We modify the scheme S in the following way to produce a scheme S'':

- $G''(1^k)$ works just like $G(1^k)$, and outputs the same secret key S_k, but instead of outputting the public key P_k, it outputs $\overline{P}_k = H(P_k)$ as the Signer's public key.
- Set $N = V_1(1^k, m)$. Then $\sigma''(1^k, P_k, S_k, m)$ produces a $b'(k)$-CS proof that $(\exists\sigma\exists P_k : (V_2(N, P_k, \sigma) = 1 \text{ and } H(P_k) = \overline{P}_k))$.
- $V''(1^k, P_k, m, \sigma'')$ tests whether or not σ'' is a valid CS proof.

If $H(\cdot)$ were an actual random oracle, then it would be very unlikely that any forger could compute any value other than P_k which is mapped to $\overline{P}_k$ by $H(\cdot)$. This is ideal from a security point of view. However, with a random oracle for $H(\cdot)$, we find ourselves unable to write a program for $\sigma''(\cdot,\cdot,\cdot,\cdot,\cdot)$, since the statement that it would be trying to produce a CS proof for would contain an oracle call— and hence would not be an NP statement. Indeed, this is precisely the reason we need our verifying algorithms to possess a "factoring" property. So instead of a random oracle, we use a collision-free hash function for $H(\cdot)$. We discuss this substitution and related matters more in section 7.5.1.

7.5 Practical considerations

There are two important points to be addressed about how feasible it actually is to use our construction of derived signature schemes.

7.5.1 Implementing random oracles

One difficulty is that our derived schemes require the use of random oracles. This is fine in theory, but to actually implement our schemes, we need to replace the random oracles with something a bit more concrete. In practice, in situations like this (see e.g. Fiat and Shamir [14]), a cryptographically strong hash function is usually substituted for a random oracle; a good hash function of this type should more or less appear to behave like a "random" function.

We can also use a "poly-random function" constructed from a one-way function, as is defined and constructed by Goldreich, Goldwasser, and Micali in [16]. However, to use a poly-random function in the setting of digital signatures, we must make public the "seed" of the poly-random function we use, which may mean that the function no longer appears to be random for our purposes. This would open up the possibility of new methods of forging which were impossible against random oracles. There are more thoughts and philosophy on these matters in Micali [22]; also, see Bellare and Rogaway [7] for a more general discussion of the role of random oracles in cryptography.

In any event, the basic idea is to replace the random oracles with functions from the "cryptographer's toolbox" which appear (to a computationally-bounded adversary) to be random. The hope is that the type of functions selected will behave sufficiently like random oracles that very little security (in some appropriate sense) will be lost in the transition from random oracles to "random" functions.

Digital signature schemes of the type we consider in this thesis depend not merely on "random" functions, but on *trapdoor functions*— families of one-way functions which each have some secret information associated with them, enabling them to be inverted quickly. The trapdoor function our sample scheme uses is squaring modulo a particular type of composite number. Unfortunately, this trapdoor function only gives $b(k) = O(k^{1/3} \cdot \log^{2/3} k)$ bits of security when mapping inputs of size $\theta(k)$ to outputs of size $\theta(k)$, since a circuit of size $2^{b(k)}$ can completely "break" the function by factoring a $\theta(k)$-bit integer (at best, of course, we could naïvely cross our fingers and hope to achieve $\theta(k)$ bits of security in this sense). It seems quite possible that *any* trapdoor function has this kind of security inefficiency. In a sense, the CS proof mechanism allows us to use very secure one-way functions, such as cryptographically strong hash functions, to bypass partially one of the unpleasant consequences of the insecurity present in trapdoor functions. In particular, the unpleasant consequence we are worried about here is the need to use inputs for trapdoor functions which are significantly longer than the amount of security that they provide.

The definition of derived signature schemes is actually somewhat reminiscent of how, for encryption of data, one often uses a public-key cryptosystem only to agree on a common key to use for a conventional cryptosystem, which is then used to encrypt the bulk of the message at hand. In other words, for convenient encryption and decryption, some kind of public-key mechanism is needed (just as, in our setting, some kind of trapdoor function is needed), but it would be inefficient to use it for encrypting an entire long message. Thus it is combined with some kind of "random" function technology to make the entire transaction much more efficient.

Of course, when using one-way functions to obtain short public keys, as in section 7.4, some of the above thoughts again apply. However, in this situation, our chosen function $H(\cdot)$ need not actually behave in a "random" manner; it suffices for it to be difficult to invert, so that it's infeasible for an adversary to obtain a spurious public key which also hashes to $\overline{P}_k$.

7.5.2 Asymptotics

A more serious problem (from a practical perspective) is that our results are of an asymptotic nature. That is, the improvements in signature length we produce do not evidence themselves except for "sufficiently large" security parameters. It is all too possible that the constants hidden in the "$O(\cdot)$" notation of our results are too big to make derived proof systems readily useful.

Let us do some rough calculations. Suppose we wish to use a derived signature scheme $S'_{(k,\kappa)}$ to sign a message with b bits of security. Essentially, there are two orthogonal ways that an adversary, Chet, can successfully forge a message:

1. Chet can break the original signature scheme, S_k.

2. Chet can break the κ-CS proof which was "layered" on top of S_k to make $S'_{(k,\kappa)}$.

We want to get a *lower* bound on κ. To do this, we need only worry about Chet forging via the second possibility. In addition, we need only consider forging circuits with no signing-oracle gates. Let's examine this in detail.

Assume with some minor loss of generality that $V_1(1^k, P_k, m) = m$. This means that a properly signed message in S_k consists of an instance (the message) and an NP witness (the signature).

To have b bits of security from $S_{(k,\kappa)}$, we need to be able to claim that any circuit $\mathcal{C}(\cdot)$ of size at most $2^{b/2}$ forges successfully with probability at most $2^{-b/2}$. Since a successful forgery is a valid (instance, κ-CS proof) pair, to obtain this conclusion from corollary 3, we need the inequality

$$[t(3s^2 + s + 3) + 1] \cdot 2^{-\kappa} \leq 2^{-b/2}$$

to hold for any circuit $\mathcal{C}(\cdot)$ of size at most $2^{b/2}$. This implies that

$$3s^2 t \leq 2^{\kappa} \cdot 2^{-b/2}.$$

Since each $f(\cdot)$-gate has 2κ inputs, and each $g(\cdot)$-gate has κ inputs, it follows that $s \cdot 2\kappa + t \cdot \kappa \leq \text{size}(\mathcal{C}(\cdot))$. However, there's not much more we can say to relate s and t to the size of $\mathcal{C}(\cdot)$ than this. So we more or less need to be able to say that whenever $s \cdot 2\kappa + t \cdot \kappa \leq 2^{b/2}$ holds, $3s^2 t \leq 2^{\kappa} \cdot 2^{-b/2}$ holds as well.

$3s^2 t$ is maximized, subject to the constraint that $s \cdot 2\kappa + t \cdot \kappa \leq 2^{b/2}$, when $s = t = \frac{2^{b/2}}{3\kappa}$. Thus we require that

$$\frac{2^{3b/2}}{9\kappa^3} \leq 2^{\kappa} \cdot 2^{-b/2},$$

or equivalently,

$$9\kappa^3 \cdot 2^{\kappa} \geq 2^{2b}.$$

If we are trying to obtain $b = 100$ bits of security from our signatures, this means that $\kappa \geq 175$. How long are the resulting $S'(k, \kappa)$-signatures? We remember that $\Lambda_{P,WL}(n, \kappa) > 2q\alpha\kappa^2$. Current research in complexity theory (see Bellare, Goldwasser, Lund, and Russell [6]) indicates that a value of q somewhere around 20 is sufficient to make probabilistically checkable proofs (recall that q is the number of proof bits of a probabilistically checkable proof that the Verifier examines). Even if the height of the tree we tree-hash with to create CS proofs is only $\alpha = 1$, this means that signatures are 1.2 Megabits long.

In contrast, the best factoring algorithms known at present factor k-bit numbers in expected running time approximately $\exp\left(1.9(k \ln 2)^{1/3}[\ln(k \ln 2)]^{2/3}\right)$ (see [1]). Let us assume that this means that our sample factoring-based digital signature scheme has $(2k \ln 2)^{1/3}[\ln(2k \ln 2)]^{2/3}$ bits of security (to break S_k, Chet needs to factor a $2k$-bit number). It turns out that $k = 8500$ provides more than 100 bits of security for S_k, if all our assumptions hold. In other words, 17 kilobit signatures suffice to provide 100 bits of security with the original digital signature scheme. This is less than 1/70 as long as the derived signatures we considered.

Obviously, these numbers do not speak well for derived digital signature schemes. However, times change. Ongoing progress in both hardware and software design continually decreases the time needed to factor large numbers and compute discrete logarithms [1], and since most digital signature schemes are based on one of these two problems, users are always finding it necessary to increase their security parameters. At some point, keys could become long enough that it becomes worthwhile to use derived signature schemes.

Chapter 8

Conclusion

We have proven that CS proofs satisfy a stronger, more constructive notion of soundness than was originally presented in [22]. Our proof enables us to use CS proofs as noninteractive proofs of knowledge— that is, they can be an efficient method for a Prover to convince a Verifier that it "knows" some NP fact.

We have demonstrated how to apply these proofs of knowledge to digital signature schemes. Starting with a digital signature scheme, we have created a *derived digital signature scheme* from it, in which the signer signs a message by providing a proof that it knows a signature of the message in the original signature scheme. For large enough security parameters, these new signatures can be much shorter than the original signatures. It is also possible to use this approach to create digital signature schemes with shorter public keys than would otherwise be needed to achieve a given security level.

Our method of modifying digital signature schemes is very general. Although (for demonstration purposes) we concentrated especially on a particular simple variant of the RSA signature scheme (see [26], [24], and [31]), our construction can be applied to any of a broad category of signature schemes— for example, the El-Gamal digital signature scheme [11].

In additional to our results about CS proofs of knowledge and how to apply them to digital signature schemes, we have also explicitly modelled security for both of these settings against a *non-uniform* adversary (i.e., against circuits), instead of against a more customary *uniform* adversary (i.e., a Turing machine). We feel that this is the proper way to model adversaries in most cryptographic settings, because it more accurately takes into account the attacks that an adversary might make— to successfully attack a cryptographic protocol, an adversary only needs to be able to "break" it for whatever value of security parameter is actually in use.

Bibliography

[1] L. M. Adleman. Algorithmic number theory— the complexity contribution. In *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science*, pages 88–113. IEEE, 1994.

[2] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and hardness of approximation problems. In *Proceedings of the 33rd Annual IEEE Symposium on Foundations of Computer Science*, pages 14–23. IEEE, 1992.

[3] S. Arora and S. Safra. Probabilistic checking of proofs: a new characterization of NP. In *Proceedings of the 33rd Annual IEEE Symposium on Foundations of Computer Science*, pages 2–13. IEEE, 1992.

[4] L. Babai, L. Fortnow, L. Levin, and M. Szegedy. Checking computations in polylogarithmic time. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, pages 21–31. ACM, 1991.

[5] L. Babai, L. Fortnow, and C. Lund. Non-deterministic exponential time has two-prover interactive protocols. *Computational Complexity*, 1:3–40, 1991.

[6] M. Bellare, S. Goldwasser, C. Lund, and A. Russell. Efficient probabilistically checkable proofs and applications to approximation. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 294–304. ACM, 1993.

[7] M. Bellare and P. Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security.* ACM, 1993.

[8] J. Benaloh and M. de Mare. One-way accumulators: a decentralized alternative to digital signatures. In *Advances in Cryptology— EUROCRYPT '93*, pages 274–285. Springer-Verlag, 1993.

[9] A. De Santis and G. Persiano. Zero-knowledge proofs of knowledge without interaction. In *Proceedings of the 33rd Annual IEEE Symposium on Foundations of Computer Science*, pages 427–436. IEEE, 1992.

[10] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[11] T. El-Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in Cryptology— CRYPTO '84*, pages 10–18. Springer-Verlag, 1984.

[12] U. Feige, A. Fiat, and A. Shamir. Zero-knowledge proofs of identity. *Journal of Cryptology*, 1(2):77–94, 1988.

[13] U. Feige, S. Goldwasser, L. Lovász, S. Safra, and M. Szegedy. Approximating clique is almost NP-complete. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 2–12. IEEE, 1991.

[14] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Advances in Cryptology— CRYPTO '86*, pages 186–194. Springer-Verlag, 1986.

[15] L. Fortnow, J. Rompel, and M. Sipser. On the power of multi-prover interactive protocols. In *Proceedings of the 3rd Annual IEEE Symposium on Structure in Complexity Theory*, pages 156–161. IEEE, 1988.

[16] O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *Journal of the Association for Computing Machinery*, 33(4):792–807, October 1986.

[17] S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against chosen-message attacks. *Siam Journal of Computing*, 17(2):281–308, April 1988.

[18] S. Khanna, R. Motwani, M. Sudan, and U. Vazirani. On syntactic versus computational views of approximability. In *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science*, pages 819–830. IEEE, 1994.

[19] J. Kilian. A note on efficient zero-knowledge proofs and arguments. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 723-732. ACM, 1992.

[20] C. Lund, L. Fortnow, H. Karloff, and N. Nisan. Algebraic methods for interactive proof systems. In *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 2–10. IEEE, 1990.

[21] R. C. Merkle. A certified digital signature. In *Advances in Cryptology— CRYPTO '89*, pages 218–238. Springer-Verlag, 1989.

[22] S. Micali. CS proofs. In *Proceedings of the 35th Annual IEEE Symposium on Foundations of Computer Science*, pages 436–453. IEEE, 1994.

[23] A. Polishchuk and D. A. Spielman. Nearly linear-size holographic proofs. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, pages 194–203. ACM, 1994.

[24] M. O. Rabin. Digitalized signatures and public-key functions as intractable as factorization. Technical Report TR-212, MIT, January 1979.

[25] R. L. Rivest, March 1995. Personal communication.

[26] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptography. *Communications of the ACM*, 21(2):120–126, 1978.

[27] A. Shamir. IP=PSPACE. In *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 11–15. IEEE, 1990.

[28] D. A. Spielman. *Computationally Efficient Error-Correcting Codes and Holographic Proofs.* PhD thesis, Massachusetts Institute of Technology, 1995.

[29] M. Sudan. *Efficient Checking of Polynomials and Proofs and the Hardness of Approximation Problems.* PhD thesis, University of California at Berkeley, 1992.

[30] M. Tompa and H. Woll. Random self-reducibility and zero knowledge interactive proofs of possession of information. In *Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science*, pages 472–482. IEEE, 1987.

[31] H. C. Williams. A modification of the RSA public-key cryptosystem. *IEEE Transactions on Information Theory*, IT-26(6):726–729, 1980.