ACE: The Advanced Cryptographic Engine*

Thomas Schweinberger, Victor Shoup
IBM Zurich Research Laboratory
Säumerstr. 4, 8803 Rüschlikon
Switzerland
{ths,sho}@zurich.ibm.com

August 14, 2000

Abstract

This document describes the Advanced Cryptographic Engine (*ACE*). It specifies a public key encryption scheme as well as a digital signature scheme with enough detail to ensure interoperability between different implementations. These schemes are almost as efficient as commercially used schemes, yet unlike such schemes, can be proven secure under reasonable and well-defined intractability assumptions. A concrete security analysis of both schemes is presented.

*Change log:
First draft, March 1, 2000.
Second draft, August 14, 2000: minor corrections.

Contents

1 Introduction

The Advanced Cryptographic Engine (*ACE*) is a library of software routines that implement a public key encryption scheme as well as a digital signature scheme. Since names are sometimes convenient, we call the encryption scheme "ACE Encrypt" and the signature scheme "ACE Sign." These schemes are almost as efficient as commercially used schemes, yet unlike such schemes, can be proven secure under reasonable and well-defined intractability assumptions. The schemes implemented are particular variants of the Cramer-Shoup encryption scheme [CS98] and the Cramer-Shoup signature scheme [CS99]. These variants have been finely tuned to strike a good balance between efficiency and security. The papers [CS98] and [CS99], as well as the related background papers [Sho00a], [Sho00b], and also [Sho98] are available on line at the following URL:

`http://www.zurich.ibm.com/Technology/Security/extern/ace`

In this document, we specify these schemes with enough detail to ensure interoperability between different implementations. We also present a concrete security analysis of both schemes.

Before doing this, however, we sketch the security goals that these schemes are meant to achieve, and the assumptions under which these goals are actually achieved.

2 Security goals

2.1 Provable security

One of the goals of modern cryptography is to design cryptographic primitives, such as signatures and encryption schemes, and to reason about their security. This task can be divided into three sub-tasks:

- to *define* an appropriate notion of security, including a formal model that describes how an adversary interacts with the system, and what constitutes "breaking" the system;
- to *design* cryptographic schemes;
- to *prove* the security of cryptographic schemes.

The importance of the definitional aspect cannot be overemphasized. It has taken a number of years for the "right" definitions for many cryptographic primitives to emerge, and there is still work to be done in defining security for more complex systems. Many cryptographic schemes have been "broken" only because the designers of the scheme did not anticipate certain modes of attack.

In terms of proving security, the ultimate goal would be to prove that a scheme cannot be broken—period. While this can be achieved for certain cryptographic problems, the solutions are generally quite impractical, and require a very special set of physical assumptions. We refer the reader to Maurer's survey on this area of *information-theoretic cryptography* [Mau99].

The next most ambitious goal for proving security would be to prove that a scheme can not be broken without the use of an inordinate amount of computing resources. Unfortunately, given the current state of mathematical knowledge, we cannot hope to prove the security of any scheme in this absolute sense. Rather, by a "provably secure" scheme, cryptographers usually mean security in a conditional sense, based upon "reasonable and natural" intractability assumptions, e.g., the assumption that factoring large numbers is hard. This is the sense in which we shall use the term "provably secure."

Although provable security in this conditional sense may not be as strong a notion as one would like, it is still a very powerful notion. It guarantees that there can be no "shortcuts" in breaking a cryptographic system—an adversary attempting to break the system must attack the underlying "hard" problems directly. There are several examples of cryptographic systems that have been proposed, and even deployed, only later to be broken via a "shortcut"—that is, *without* solving the underlying "hard" problem. One of the more spectacular such examples is Bleichenbacher's chosen ciphertext attack on RSA's encryption scheme, PKCS #1 [Ble98]. Even though the underlying encryption scheme is based on the RSA problem (see §2.6), Bleichenbacher's attack cleverly breaks the scheme without solving this problem. This attack rendered insecure the widely deployed SSL key agreement protocol, which is based on this encryption scheme. Another recent example is an attack on the ISO 9761-1 standard for digital signatures [CNS99, CHJ99]. Again, even though the scheme is based on the RSA problem, the attack cleverly breaks the scheme without solving this problem.

Random oracle arguments

There are a number of examples in the literature of cryptographic schemes that are either provably secure but hopelessly impractical, or practical but lacking a proof of security (or even broken). Schemes that are both truly practical *and* provably secure are hard to come by. Because of this, a new trend has emerged in the cryptographic research community: proofs of security in an idealized model of computation wherein a cryptographic hash function (like MD5 or SHA-1) is treated *as if* it were a *random oracle*, i.e., a "black box" that contains a random function which can only be evaluated by making an explicit query. This "random oracle" model for security analysis was informally introduced by [FS87], and later formalized by [BR93]. It has been used to analyze numerous cryptographic systems (see, e.g., [BR94] and [PS96]). However, we must emphasize that making use of random oracles is *not* just another assumption—a cryptographic hash function is not, and never can be, a random oracle. It is entirely possible that a cryptographic scheme that is secure in the random oracle model can be broken without either breaking the underlying hard problem, or finding any particular weakness in the cryptographic hash function. Indeed, this is amply demonstrated in [CGH98]. Our point of view is that a security analysis in the random oracle model is best viewed as *heuristic evidence* for the security of a scheme. If the only practical solutions to a problem rely on a random oracle argument for their proof of security, fine—this is much better than no security analysis at all; but if a practical solution can be obtained without relying on a random oracle argument, so much the better.

Shortly after RSA's PKCS #1 was shown to be vulnerable to a chosen ciphertext attack, it was modified so as to utilize Bellare and Rogaway's OAEP encryption scheme [BR94]. This scheme is provably secure in the random oracle model (assuming the

RSA problem is hard). The encryption scheme described in this document is provably secure—without making random oracle arguments—and is not too much less efficient than OAEP. Although there may be scenarios where the engineering requirements are so constraining that even this slight loss of efficiency cannot be tolerated, we believe that there are other scenarios where this tradeoff between efficiency and provably security is certainly worth making.

Choosing intractability assumptions

The notion of "provable security" is not entirely precise, since one has a certain flexibility in choosing the "reasonable and natural" intracatbility assumptions on which a proof of security can be based.

There are several characteristics that are desirable in an intractability assumption. Ideally, the "hard" problem should be well studied. Failing that, the problem should at least be fairly natural and easy to describe, so that it can be understood and studied by a reasonable number of people. At the very least, we believe that the problem should be *non-interactive*, that is, of the form: given an instance of a problem (e.g., the product of two large, random primes), it is hard to solve the problem (e.g., factor the number). The reason for this is that cryptographic primitives and protocols can be attacked in quite complicated and subtle ways by an adversary that interacts with the system, and such interaction is quite subtle to analyze. Reducing the security of a complex, interactive system to the hardness of a non-interactive problem can be seen as one of the main activities of modern theoretical cryptography. Another nice feature of requiring non-interactive assumptions is that it rules out the "proof technique" of proving a cryptosystem is secure by assuming *a priori* that it is secure.

The reason we spend some time discussing what we believe constitutes a "reasonable and natural" intractability assumption is that some researchers apparently have a much more liberal interpretation of the term. For example, in [ZS92], the authors prove the security of an encryption scheme based on an assumption of the form: an arbitrary adversary can be replaced by an essentially equivalent adversary that behaves in a certain nice way. As one can see, by our standards, this is not a reasonable intractability assumption—it is really just a proof of security against a restricted class of adversaries. As another example, in [ABR98], the authors make intractability assumptions that are *interactive*; indeed, these intractability assumptions amount to little more than a restatement of the definition of security in terms of the particular implementation that they propose. We believe this misses the whole point of "provable security," and it certainly does not meet our standard of a reasonable intractability assumption.

2.2 Secure public key encryption

The development of a practically useful and mathematically meaningful definition of secure public key encryption took the cryptographic research community a number of years. There are a number of weak, *ad hoc*, notions of security which are *not* very useful. These include (1) the requirement that the private key should be hard to recover, and (2) the requirement that individual ciphertexts should be hard to decrypt.

The first step towards a workable definition was the formulation of the notion of *semantic security* by [GM84]. This definition of security captures the notion that a ciphertext leaks no information about the corresponding cleartext to a (computationally bounded) eavesdropper.

We sketch this definition in more detail. Briefly, security in this sense means that it is infeasible for an adversary to gain a non-negligible advantage in the following game. A public key/private key pair for the scheme is generated, and the adversary is given the public key. Then the adversary generates two equal length messages m_0, m_1, and gives these to an *encryption oracle*. We assume these two messages have non-zero length.[1] The encryption oracle chooses a bit $b \in \{0, 1\}$ at random, encrypts m_b, and gives the adversary the corresponding *target* ciphertext ψ'. Finally, the adversary outputs his guess at b. The adversary's advantage is defined to be the distance from 1/2 of the probability that his guess is correct.

As mentioned above, the formal definition of semantic security captures the intuitive notion that no information about an encrypted message is leaked to a *passive* adversary that only eavesdrops. In protocol design and analysis, a much more robust definition is often required that captures the intuitive notion of security against an *active* attack, in which the adversary not only can eavesdrop, but can inject his own messages into the network. The type of security one needs in this setting is *non-malleability*, also called *security against chosen ciphertext attack*, a notion that was formalized in the sequence of papers [NY90, RS91, DDN91].

The definition of non-malleability is the same as for semantic security, but with the following essential difference. The adversary is given access to a *decryption oracle* throughout the entire game; the adversary may request the decryption of ciphertexts ψ of his choosing, subject only to the (obviously necessary) restriction that after the target ciphertext ψ' has been generated, the adversary may not request the decryption of ψ' itself.

Another intuitive way to understand non-malleability (and the motivation for its name) is that a non-malleable encryption scheme essentially provides a *secure envelope*, that is, an envelope whose contents can neither be seen nor modified by an adversary.

Non-malleability is a fundamental notion that is necessary to ensure the security of numerous protocols that use public-key encryption. Sometimes, security engineers appear to implicitly assume that a given encryption scheme is non-malleable, even if there is no justification for this. A case in point is Bleichenbacher's attack on SSL (see §2.1).

For further discussion on the importance of non-malleability, see [Sho98].

The above definitions for semantic security and non-malleability may seem somewhat limited at first sight—in particular, one might ask what security properties are guaranteed in a richer attack scenario where there are many users with public keys and many messages are encrypted under these public keys. However, the above definitions are quite robust, and it is well known that they are essentially equivalent to just about any reasonable generalization one might consider in a multi-user/multi-message environment. For a detailed account of this issue, see [BBM00].

[1] A user might encrypt a zero length message, but this is not interesting from a security point of view.

Concrete security analysis

In this document, we want to carry out a concrete (or exact) exact security analysis. That is, we want to develop an *explicit*, quantitative relationship between the hardness of breaking a cryptosystem and the hardness of the underlying problems on which it is based. In order to facilitate this, we define

$$\mathsf{AdvEnc}(t, \kappa, l)$$

to be the advantage in the above game defining non-malleability, where we consider an adversary that runs in time at most t, makes at most κ decryption requests, and l is an upper bound on the length (in bytes, say) of the test messages m_0, m_1.

This function implicitly depends on the security parameters chosen to define the signature scheme.

Also note that this function depends on the *model of computation*, since the notion of "time" depends on the details of this model. We do not want to get mired in the details of this. A perfectly acceptable model is to fix a simple stored-program machine model with a fixed word size (32 or 64 bits) and a convenient and realistic instruction set, and then to measure time by counting the number of instructions executed. We also count in the running time the size of the program, as well as any pre-initialized data tables.

Note that for simplicity, in the adversary's time we count the time spent by the key generation algorithm, encryption algorithm, and decryption algorithm—that is, the entire running time of the attack game is "charged" to the adversary. Also, we shall view t as a strict bound on the running time, and not, say, an expected value.

2.3 Secure digital signatures

The notion of security we want is that of security against existential forgery against adaptive chosen message attack, as defined in [GMR88]. This is the strongest, and most useful notion of security, allowing a signature scheme to be used in an arbitrary application without restrictions.

Briefly, security in this sense means that it is infeasible for an adversary to win the following game. A public key/private key for the scheme is generated, and the adversary is given the public key. The adversary then makes a sequence of signing requests. The messages for which the adversary requests signatures can be adaptively chosen, i.e., they may depend on previous signatures. The adversary wins the game if he can forge a signature, i.e., can output a message other than one for which he requested a signature, along with a valid signature on that message.

Concrete security analysis

In order to facilitate concrete security analysis, we define

$$\mathsf{AdvSig}(t, \kappa, l)$$

to be the probability that an adversary wins the above game, where we consider adversaries that run in time at most t, make at most κ signing requests, and l is an upper bound on the total length (in bytes, say) of all the signed messages.

All of the technical caveats on the definition of AdvEnc apply to the definition of AdvSig as well.

2.4 Intractability assumptions

The signature scheme and encryption scheme in ACE can be proven secure under reasonable and natural intractability assumptions, without resorting to random oracle arguments. However, we do make use of cryptographic hash functions as a "hedge": in the random oracle model, the schemes in ACE can be proven secure under even weaker intractability assumptions.

The four basic assumptions we need are as follows:

(1) The Decisional Diffie-Hellman (DDH) assumption.

(2) The Strong RSA assumption.

(3) SHA-1 second preimage collision resistance.

(4) MARS sum/counter mode pseudo-randomness.

We need assumptions (1), (3), and (4) to prove the security of the encryption scheme, and we need assumptions (2), (3), and (4) to prove the security of the signature scheme. In the random oracle model, assumptions (1) and (2) can be replaced by

(1′) The Computational Diffie-Hellman (CDH) assumption.

(2′) The RSA assumption.

Thus, although we need to make somewhat strong intractability assumptions to get a true proof of security, our schemes are in a sense no less secure than more traditional schemes that are based on assumptions (1′) and (2′), but which (at best) can be analyzed *only* in the random oracle model.

We now describe these assumptions in some detail.

2.5 The Computational and Decisional Diffie-Hellman assumption

Let G be a group of large prime order q and let $g \in G$ be a generator. The Computational Diffie-Hellman (CDH) assumption, introduced by [DH76], is the assumption that computing g^{xy} from g^x and g^y is hard. It is a widely held belief that the security of certain key exchange protocols (such as STS [DvOW92]) is implied by the CDH assumption. This is simply false—under *any* reasonable definition of security—except in the random oracle model of security analysis. What is almost always needed, but often not explicitly stated, is the Decisional Diffie-Hellman (DDH) assumption.

For $g_1, g_2, u_1, u_2 \in G$, define $DHP(g_1, g_2, u_1, u_2)$ to be 1 if there exists $x \in \mathbf{Z}_q$ such that $u_1 = g_1^x$ and $u_2 = g_2^x$, and 0 otherwise. A "good" algorithm for DHP is an efficient, probabilistic algorithm that computes DHP correctly with negligible error probability *on all inputs*. The DDH assumption is the assumption that there is no good algorithm for DHP.

This formulation is equivalent to the more usual one where

$$g_1 = g, g_2 = g^x, u_1 = g^y, u_2 = g^{xy}.$$

The DDH assumption is a potentially stronger assumption than the CDH assumption, but at the present time, the only known method for breaking either assumption is to solve the Discrete Logarithm problem.

The DDH assumption appears to have first surfaced in the cryptographic literature in a paper by S. Brands [Bra93]. See [Bon98, CS98, NR97, Sta96] for further applications of and discussions about the DDH assumption.

The groups G that are used in *ACE* are prime-order subgroups of the multiplicative group of units modulo a large prime. These subgroups have order roughly 2^{256}.

Random self reduction and an equivalent formulation of the DDH

There are a few useful random self-reductions that allow us to transform arbitrary inputs to *DHP* into random inputs on which *DHP* evaluates to the same value.

Let g_1, g_2, u_1, u_2 be given such that $g_1 \neq 1$ and $g_2 \neq 1$. We can randomize u_1 and u_2 as follows:

$$\tilde{u}_1 = u_1^a g_1^b,\ \tilde{u}_2 = u_2^a g_2^b,$$

where $a, b \in \mathbf{Z}_q$ are chosen at random. Suppose that $u_1 = g_1^x$ and $u_2 = g_2^y$. If $x = y$, then $(\tilde{u}_1, \tilde{u}_2)$ is a random pair of group elements, subject to $\log_{g_1}(\tilde{u}_1) = \log_{g_2}(\tilde{u}_2)$. If $x \neq y$, then $(\tilde{u}_1, \tilde{u}_2)$ is a pair of random, independent group elements.

Next, we can randomize g_2 as follows:

$$\tilde{g}_2 = g_2^c,\ \tilde{u}_1 = u_1^a g_1^b,\ \tilde{u}_2 = u_2^{ac} g_2^{bc},$$

where $c \in \mathbf{Z}_q$ is chosen at random.

Additionally, we can randomize g_1 as follows:

$$\tilde{g}_1 = g_1^d,\ \tilde{g}_2 = g_2^c,\ \tilde{u}_1 = u_1^{ad} g_1^{bd},\ \tilde{u}_2 = u_2^{ac} g_2^{bc},$$

where $d \in \mathbf{Z}_q$ is chosen at random.

With this transformation, we see that we can transform an arbitrary input to *DHP* to an equivalent, random input. From this, it follows that the two distributions

$$\mathbf{R}:\ (g_1, g_2, g_1^x, g_2^y),\ \text{random } g_1, g_2 \in G\backslash\{1\};\ x, y \in \mathbf{Z}_q,$$

and

$$\mathbf{D}:\ (g_1, g_2, g_1^x, g_2^x),\ \text{random } g_1, g_2 \in G\backslash\{1\};\ x \in \mathbf{Z}_q$$

are computationally indistinguishable under the DDH assumption. This random self-reducibility property was first observed by Stadler [Sta96] (and also independently in [NR97]).

Concrete security analysis

In order to facilitate concrete security analysis, we define

$$\mathsf{AdvDDH}(t)$$

to be the maximum over all statistical tests T that run in time at most t and output $0, 1$ of

$$\left| \Pr[T(\mathbf{R}) = 1] - \Pr[T(\mathbf{D}) = 1] \right|.$$

2.6 The RSA and strong RSA assumptions

The RSA problem is the following. Given a randomly generated RSA modulus n, an exponent r, and a random $z \in \mathbf{Z}_n^*$, find $y \in \mathbf{Z}_n^*$ such that $y^r = z$. The exponent r is drawn from a particular distribution—particular distributions give rise to particular versions of the RSA problem. The *RSA assumption* is the assumption that this problem is hard to solve.

The *flexible* RSA problem is the following. Given an RSA modulus n and a random $z \in \mathbf{Z}_n^*$, find $r > 1$ and $y \in \mathbf{Z}_n^*$ such that $y^r = z$. The choice of r may be restricted in some fashion—particular restrictions give rise to particular versions of the flexible RSA problem. The *strong RSA assumption* is the assumption that this problem is hard to solve. Note that this differs from the ordinary RSA assumption, in that for the RSA assumption, the exponent r is chosen independently of z, whereas for the strong RSA assumption, r may be chosen in a way that depends on z. The strong RSA assumption is a potentially stronger assumption than the RSA assumption, but at the present time, the only known method for breaking either assumption is to solve the integer factorization problem.

The strong RSA assumption was introduced in [BP97], and has subsequently been used in the analysis of several cryptographic schemes (see, e.g., [FO99, GHR99]).

Concrete security analysis

We define

$$\mathsf{AdvRSA}(t)$$

to be the maximum over all algorithms that run in time at most t of the probability of solving the RSA problem. We also define

$$\mathsf{AdvFlexRSA}(t)$$

to be the corresponding probability for solving the flexible RSA problem.

Random Self Reduction

One of the nice features about the RSA problem is that it is random self-reducible. That is, having fixed n and r, then the problem of computing $y = z^{1/r}$ for an *arbitrary* $z \in \mathbf{Z}_n^*$ can be reduced to the problem of computing $\tilde{y} = \tilde{z}^{1/r}$ for *random* $\tilde{z} \in \mathbf{Z}_n^*$. This means that given an efficient algorithm to solve the latter problem, one can efficiently solve the former problem. This is a well-known and quite trivial reduction: given z, choose $s \in \mathbf{Z}_n^*$ at random, and set $\tilde{z} = s^r z$. Then we have $y = \tilde{y}/s$.

The existence of such a random self reduction adds credibility to the RSA assumption, since if there is an algorithm that solves the RSA problem for a given n and for a non-negligible fraction of choices of z, then there is another algorithm that solves the RSA problem for the same n for all choices of z.

There is also a random self reduction for the flexible RSA problem, at least in the particular version that we need to prove the security of the signature scheme. Just as for the RSA problem, this random self reduction adds credibility to the strong RSA assumption. This reduction appears not to be so well known, and is described in detail in the full-length version of [CS99].

2.7 SHA-1 second preimage collision resistance

The notion of a UOWHF was introduced by Naor and Yung [NY89]. A UOWHF is a keyed hash function with the following property: if an adversary chooses a message x, and then a key K is chosen at random and given to the adversary, it is hard for he adversary to find a different message $x' \neq x$ such that $H_K(x) = H_K(x')$.

As a cryptographic primitive, a UOWHF is an attractive alternative to the more traditional notion of a *collision-resistant hash function* (CRHF), which is characterized by the following property: given a random key K, it is hard to find two different messages x, x' such that $H_K(x) = H_K(x')$.

A UOWHF is an attractive alternative to a CRHF because

(1) it seems easier to build an efficient and secure UOWHF than to build an efficient and secure CRHF, and

(2) in many applications, most importantly for building digital signature schemes, a UOWHF is sufficient.

As evidence for claim (1), we point out the recent attacks on MD5 [dBB93, Dob96]. We also point out the complexity-theoretic result of Simon [Sim98] that shows that there exists an oracle relative to which UOWHFs exist but CRHFs do not. CRHFs can be constructed based on the hardness of specific number-theoretic problems, like the discrete logarithm problem [Dam87]. Simon's result is strong evidence that CRHFs cannot be constructed based on an arbitrary one-way permutation, whereas Naor and Yung [NY89] show that a UOWHF can be so constructed.

As we shall see, *ACE* needs only a UOWHF. We construct such a UOWHF by using the composition theorem in [Sho00a], together with the SHA-1 low-level compression function

$$C : \{0,1\}^{672} \to \{0,1\}^{160}$$

as the basic primitive. The assumption we make about C is that it is *second preimage collision resistant*, i.e., if a random input $x \in \{0,1\}^{672}$ is chosen, then it is hard to find different input $x' \neq x$ such that $C(x) = C(x')$. This assumption seems to be much weaker than assumption that no collisions in C can be found at all (which as an intractability assumption does not even make sense). Indeed, the techniques used to find collisions in MD5 [dBB93, Dob96] do not appear to help in finding second preimages.

Note that from a complexity theoretic point of view, second preimage collision resistance is no stronger than the UOW property. Indeed, if $H_K(x)$ is a UOWHF, then the function sending (K, x) to $(K, H_K(x))$ is second preimage collision resistant.

All of the above tends to indicate that the assumption that C is second preimage collision resistant is much more reasonable than the assumption that C is collision resistant. Also note that from a concrete, quantitative security point of view, second preimage collision resistance is also quite attractive. The SHA-1 compression function C has a 160-bit output. Because of the birthday paradox, collisions can be found by brute-force search in 2^{80} steps, but a brute-force search for a second preimage would require 2^{160} steps. In not too many years, an attack that takes 2^{80} steps may be near the threshold of feasibility; in this situation, a scheme that relies on the collision resistance for C can no longer be considered secure, whereas a scheme that relies only on second preimage collision resistance may still be considered secure, provided no attack substantially better than a brute-force attack is discovered.

Concrete security analysis

We define

$$\mathsf{AdvSHA}(t)$$

to be the maximum over all algorithms that run in time at most t of the probability of finding second preimages for SHA-1, as defined above.

2.8 MARS sum/counter mode pseudo-randomness

We will make use of the MARS block cipher [BCD+98] in sum/counter mode to generate sequences of pseudo-random bits.

Let $f(k, x)$ denote the evaluation of the block cipher MARS using a 256-bit key k and a 128-bit input block x, yielding a 128-bit output block. The assumption we make about MARS is that when used in sum/counter mode, the resulting sequence of bits is pseudo-random.

More precisely, consider the following two distributions, for a given length parameter $l > 0$:

$$\mathbf{P}_l : \ (x, f(k, x) \oplus f(k, x + 1), \ldots, f(k, x + 2l - 2) \oplus f(k, x + 2l - 1)),$$

where k is a random 256-bit string and x is a random 128-bit string, and

$$\mathbf{R}_l : \ (x, r_0, \ldots, r_{l-1}),$$

where $x, r_0, \ldots, r_{l-1}$ are random 128-bit strings. Here, we interpret "$x + j$," for $0 \le j < 2l$ in the natural way as the 128-bit block representing $x + j$ reduced modulo 2^{128}.

The pseudo-randomness assumption we make is that the two distributions $\mathbf{P}_l$ and $\mathbf{R}_l$ are computationally indistinguishable.

Concrete security analysis

In order to facilitate concrete security analysis, we define

$$\mathsf{AdvMARS}(t, l)$$

to be the maximum over all statistical tests T that run in time at most t and output $0, 1$ of

$$\left| \Pr[T(\mathbf{R}_l) = 1] - \Pr[T(\mathbf{P}_l) = 1] \right|.$$

Summed MARS

Note that in this construction, instead of using the output of MARS in counter mode directly, we take the *exclusive or* of consecutive pairs of MARS outputs. This of course degrades the speed by a factor of two, but there are some advantages from a security pont of view.

First, since the adversary does not see any MARS input/output pairs, but only the *exclusive ors* of outputs, certain types of cryptanalysis should be less feasible.

Second, and more important, this construction goes a long way to hiding the fact that MARS actually behaves like a random permutation, and not a random function. Indeed, if we just use MARS directly in counter mode, then we can distinguish its output from random with an advantage close to $l^2/2^{128}$, simply because in a sequence of random blocks, we would expect a collision, but none is forthcoming from MARS. Recent results of [BI99] imply that the sum/counter mode construction reduces the advantage to something much closer to $l/2^{128}$. A similar result has also been independently obtained by [Luc00]. The latter result is based on a much more elementary proof, and is somewhat weaker; however, for $l < 2^{64}$, the result in [Luc00] is nearly as good as that in [BI99].

3 Terminology and Notation

In order to describe the encryption and signature schemes precisely, we need to establish some notational conventions.

3.1 Basic mathematical notation

- $\mathbf{Z}$

  The set $\mathbf{Z}$ of integers.

- $\mathbf{F}_2[T]$

  The set $\mathbf{F}_2[T]$ of univariate polynomials with coefficients in the finite field $\mathbf{F}_2$ of cardinality 2.

- $A \operatorname{rem} n$

  For $A \in \mathbf{Z}$ and integer $n > 0$, then $A \operatorname{rem} n$ is defined to be the integer $r \in \{0, \ldots, n-1\}$ such that $A \equiv r \pmod{n}$.

- $A \operatorname{rem} f$

  For $A, f \in \mathbf{F}_2[T]$ with $f \neq 0$, $A \operatorname{rem} f$ is defined to be the polynomial $r \in \mathbf{F}_2[T]$ with $\deg(r) < \deg(f)$ such that $A \equiv r \pmod{f}$.

3.2 Basic string notation

Fix a set A. A^* denotes the set of all strings, i.e., finite sequences, over the set A. For $n \geq 0$, A^n denotes the set of all sequences of length n over A.

For a string $x \in A^*$, $L(x)$ denotes its length. The string of length zero is denoted λ_A.

Let $x = (a_0, \ldots, a_{m-1}) \in A^m$ be a string of length m, where $a_i \in A$ for $0 \leq i < m$. For $0 \leq i \leq j \leq m$, we define the *substring* operation

$$[x]_i^j \stackrel{\text{def}}{=} (a_i, \ldots, a_{j-1}) \in A^{j-i}.$$

For $0 \leq i \leq m-1$, we define the *selection* operation

$$x[i] \stackrel{\text{def}}{=} a_i \in A.$$

For $x, y \in A^*$, we define $z = x \,\|\, y$ to be the *concatenation* of x and y. That is, $z \in A^*$ is the unique string such that $L(z) = L(x) + L(y)$, $[z]_0^{L(x)} = x$, and $[z]_{L(x)}^{L(z)} = y$.

3.3 Bits, bytes, and words

Define $\mathbf{b} \stackrel{\text{def}}{=} \{0, 1\}$, the set of *bits*. We will work with sets of the form

$$\mathbf{b},\ \mathbf{b}^{n_1},\ (\mathbf{b}^{n_1})^{n_2},\ \ldots.$$

For such a set A, we define the "zero element" $0_A \in A$ recursively, as follows:

$$\begin{aligned} 0_{\mathbf{b}} &\stackrel{\text{def}}{=} 0 \in \mathbf{b}; \\ 0_{A^n} &\stackrel{\text{def}}{=} (0_A, \ldots, 0_A) \in A^n \quad \text{for } n \geq 0. \end{aligned}$$

We define $\mathbf{B} \stackrel{\text{def}}{=} \mathbf{b}^8$, the set of *bytes*.

We define $\mathbf{W} \stackrel{\text{def}}{=} \mathbf{b}^{32}$, the set of *words*.

For $x \in A^*$ with $A \in \{\mathbf{b}, \mathbf{B}, \mathbf{W}\}$, and for $l \geq 0$, we define a padding operator

$$pad_l(x) \stackrel{\text{def}}{=} \begin{cases} x & \text{if } L(x) \geq l; \\ x \,\|\, 0_{A^{l-L(x)}} & \text{otherwise.} \end{cases}$$

For $x \in A^*$ with $A \in \{\mathbf{b}, \mathbf{B}, \mathbf{W}\}$, we say that x is *normalized* if x is not of the form $y \,\|\, 0_{A^n}$ for some $y \in A^*$ and some $n > 0$.

3.4 Conversion operators

We define a number of conversions among $\mathbf{Z}, \mathbf{F}_2[T], \mathbf{b}^*, \mathbf{B}^*, \mathbf{W}^*$. The general notation for a conversion operator is

$$I_{src}^{dst} : src \to dst,$$

which is a function that converts an element of the set *src* to an element of the set *dst*.

All of these conversion operators are quite simple and natural, even though their formal specification is a little tedious. The only thing to really notice is that the conversion between byte strings and word strings follows what is sometimes called the "little endian" ordering convention.

- $I^{\mathbf{Z}}_{\mathbf{b}^*}(x) \stackrel{\text{def}}{=} \sum_{i=0}^{L(x)-1} x[i]2^i$.
- $I^{\mathbf{b}^*}_{\mathbf{Z}}(n) \stackrel{\text{def}}{=} x$, where $x \in \mathbf{b}^*$ is the unique, normalized bit string such that $I^{\mathbf{Z}}_{\mathbf{b}^*}(x) = |n|$.
- $I^{\mathbf{F}_2[T]}_{\mathbf{b}^*}(x) \stackrel{\text{def}}{=} \sum_{i=0}^{L(x)-1} x[i]T^i$.
- $I^{\mathbf{b}^*}_{\mathbf{F}_2[T]}(f) \stackrel{\text{def}}{=} x$, where $x \in \mathbf{b}^*$ is the unique, normalized bit string such that $I^{\mathbf{F}_2[T]}_{\mathbf{b}^*}(x) = f$.
- $I^{\mathbf{b}^*}_{\mathbf{B}^*}(x) \stackrel{\text{def}}{=} x[0] \,\|\, x[1] \,\|\, \cdots \,\|\, x[L(x)-1]$.
- $I^{\mathbf{B}^*}_{\mathbf{b}^*}(x) \stackrel{\text{def}}{=} y$, where $y \in \mathbf{B}^*$ is the unique byte string with $L(y) = \lceil L(x)/8 \rceil$ and $I^{\mathbf{b}^*}_{\mathbf{B}^*}(y) = pad_{8L(y)}(x)$.
- $I^{\mathbf{b}^*}_{\mathbf{W}^*}(x) \stackrel{\text{def}}{=} x[0] \,\|\, x[1] \,\|\, \cdots \,\|\, x[L(x)-1]$.
- $I^{\mathbf{W}^*}_{\mathbf{b}^*}(x) \stackrel{\text{def}}{=} y$, where $y \in \mathbf{W}^*$ is the unique word string with $L(y) = \lceil L(x)/32 \rceil$ and $I^{\mathbf{b}^*}_{\mathbf{W}^*}(y) = pad_{32L(y)}(x)$.
- $I^{\mathbf{B}^*}_{\mathbf{W}^*}(x) \stackrel{\text{def}}{=} y$, where $y \in \mathbf{B}^*$ is the unique byte string such that $I^{\mathbf{b}^*}_{\mathbf{B}^*}(y) = I^{\mathbf{b}^*}_{\mathbf{W}^*}(x)$.
- $I^{\mathbf{W}^*}_{\mathbf{B}^*}(x) \stackrel{\text{def}}{=} y$, where $y \in \mathbf{W}^*$ is the unique word string with $L(y) = \lceil L(x)/4 \rceil$ and $I^{\mathbf{B}^*}_{\mathbf{W}^*}(y) = pad_{4L(y)}(x)$.
- $I^{\mathbf{Z}}_{\mathbf{B}^*}(x) \stackrel{\text{def}}{=} I^{\mathbf{Z}}_{\mathbf{b}^*}(I^{\mathbf{b}^*}_{\mathbf{B}^*}(x))$
- $I^{\mathbf{B}^*}_{\mathbf{Z}}(n) \stackrel{\text{def}}{=} I^{\mathbf{B}^*}_{\mathbf{b}^*}(I^{\mathbf{b}^*}_{\mathbf{Z}}(n))$
- $I^{\mathbf{F}_2[T]}_{\mathbf{B}^*}(x) \stackrel{\text{def}}{=} I^{\mathbf{F}_2[T]}_{\mathbf{b}^*}(I^{\mathbf{b}^*}_{\mathbf{B}^*}(x))$
- $I^{\mathbf{B}^*}_{\mathbf{F}_2[T]}(f) \stackrel{\text{def}}{=} I^{\mathbf{B}^*}_{\mathbf{b}^*}(I^{\mathbf{b}^*}_{\mathbf{F}_2[T]}(f))$
- $I^{\mathbf{Z}}_{\mathbf{W}^*}(x) \stackrel{\text{def}}{=} I^{\mathbf{Z}}_{\mathbf{b}^*}(I^{\mathbf{b}^*}_{\mathbf{W}^*}(x))$
- $I^{\mathbf{W}^*}_{\mathbf{Z}}(n) \stackrel{\text{def}}{=} I^{\mathbf{W}^*}_{\mathbf{b}^*}(I^{\mathbf{b}^*}_{\mathbf{Z}}(n))$
- $I^{\mathbf{F}_2[T]}_{\mathbf{W}^*}(x) \stackrel{\text{def}}{=} I^{\mathbf{F}_2[T]}_{\mathbf{b}^*}(I^{\mathbf{b}^*}_{\mathbf{W}^*}(x))$

3.5 Other operators

For $x, y \in \mathbf{b}$, we define $z = x \oplus y \in \mathbf{b}$ to be the *exclusive-or* of x and y, i.e., $z = (x + y) \text{ rem } 2$. We can extend the $\oplus$ operator element-wise to equal-length bit strings. This defines an $\oplus$ operator on $\mathbf{B}$ and $\mathbf{W}$, which we can then extend to equal-length byte and word strings.

For convenience, for $n \in \mathbf{Z}$, we define

$$\begin{aligned} L_{\mathbf{b}}(n) &\stackrel{\text{def}}{=} L(I^{\mathbf{b}^*}_{\mathbf{Z}}(n)), \\ L_{\mathbf{B}}(n) &\stackrel{\text{def}}{=} L(I^{\mathbf{B}^*}_{\mathbf{Z}}(n)), \\ L_{\mathbf{W}}(n) &\stackrel{\text{def}}{=} L(I^{\mathbf{W}^*}_{\mathbf{Z}}(n)). \end{aligned}$$

It will also be convenient to define a simple "increment" operator on word strings. Let $x \in \mathbf{W}^n$ for some $n > 0$. Then

$$x + 1 \stackrel{\text{def}}{=} pad_n(I_{\mathbf{Z}}^{\mathbf{W}^*}((I_{\mathbf{W}^*}^{\mathbf{Z}}(x) + 1) \text{ rem } 2^{32n})) \in \mathbf{W}^n.$$

We will make use of the following low-level cryptographic transformations:

- *MARS*

  MARS encryption function as specified in [BCD+98], used with 256-bit keys; that is

  $$MARS : \mathbf{W}^8 \times \mathbf{W}^4 \to \mathbf{W}^4,$$

  where an input (k, m) consists of the key k and the input block m to be encrypted, and the output is the resulting encrypted block; we do not make use of the corresponding decryption function.

- *CSHA1*

  SHA-1 core compression function as described in [SHA95]; that is

  $$CSHA1 : \mathbf{W}^5 \times \mathbf{W}^{16} \to \mathbf{W}^5,$$

  where an input (h, m) consists of the initial hash state h and a text input m, and the output is the resulting final hash state.

3.6 Algorithmic notation

We use a fairly standard notation for describing algorithms. We use the notation $A \leftarrow B$ to denote the action of assigning the value of B to the variable A. All of our algorithms are written as "pure" functions that take an input and return an output using a "return" statement, and do not have any "side effects." Some functions may return one of several symbolic values ($\mathsf{Accept}, \mathsf{Reject}, \mathsf{Prime}, \mathsf{Composite}$).

Random numbers

At some points in the description of algorithms, we say something like "generate a random such and such." To implement this, one would need access to a source of true random bits. However, most implementations will not have access to such a source. Instead, it is presumed that a pseudo-random source is used. In all cases, the implementor should use a *cryptographically strong* source of pseudo-random bits or numbers, and ensure that the constructed objects have distributions as close as possible to truly random objects.

An implementation tip

When we describe algorithms, there are several places where conversions are performed between byte and word strings. In a careful implementation, one should convert all byte strings to word strings as early as possible, and thereafter work exclusively with word strings, since all the low-level operations work directly on words, not bytes.

4 Encryption Scheme

This section defines the public key encryption scheme. It is a variant of the hybrid version of [CS98] described in [Sho00b].

4.1 Encryption Key Pair

The encryption scheme defined in this document employs two key types, whose representation consists of the following tuples:

ACE Encryption public key: $(P, q, g_1, g_2, c, d, h_1, h_2, k_1, k_2)$.
ACE Encryption private key: (w, x, y, z_1, z_2).

For a given size parameter m, with $1024 \le m \le 16,384$, the components are as follows:

q – a 256-bit prime number.

P – an m-bit prime number with $P \equiv 1 \pmod{q}$.

g_1, g_2, c, d, h_1, h_2 – elements of $\{1, \dots, P-1\}$ (whose multiplicative order modulo P divides q).

w, x, y, z_1, z_2 – elements of $\{0, \dots, q-1\}$.

k_1, k_2 – elements of $\mathbf{B}^*$, with $L(k_1) = 20l' + 64$ and $L(k_2) = 32\lceil l/16 \rceil + 40$, where $l = \lceil m/8 \rceil$ and $l' = L_{\mathbf{b}}(\lceil (2\lceil l/4 \rceil + 4)/16 \rceil)$.

4.2 Key Generation

Algorithm 4.2.1 generates an *ACE* encryption key pair.

Algorithm 4.2.1 *Key generation for the ACE public-key encryption scheme.*

Input: A size parameter $1024 \le m \le 16,384$.

Output: A public key/private key pair, as described in §4.1.

1. Generate a random prime q, where $2^{255} < q < 2^{256}$.
2. Generate a random prime P, $2^{m-1} < P < 2^m$, such that $P \equiv 1 \pmod{q}$.
3. Generate a random integer $g_1 \in \{2, \dots, P-1\}$ such that ${g_1}^q \equiv 1 \pmod{P}$.
4. Generate random integers $w \in \{1, \dots, q-1\}$ and $x, y, z_1, z_2 \in \{0, \dots, q-1\}$.
5. Compute the following integers in $\{1, \dots, P-1\}$:

$$\begin{aligned} g_2 &\leftarrow {g_1}^w \text{ rem } P, \\ c &\leftarrow {g_1}^x \text{ rem } P, \\ d &\leftarrow {g_1}^y \text{ rem } P, \\ h_1 &\leftarrow {g_1}^{z_1} \text{ rem } P, \\ h_2 &\leftarrow {g_1}^{z_2} \text{ rem } P. \end{aligned}$$

6. Generate random byte strings $k_1 \in \mathbf{B}^{20l'+64}$, and $k_2 \in \mathbf{B}^{32\lceil l/16\rceil+40}$, where $l = L_{\mathbf{B}}(P)$ and $l' = L_{\mathbf{b}}(\lceil(2\lceil l/4\rceil + 4)/16\rceil)$.

7. Return the public key/private key pair

$$((P, q, g_1, g_2, c, d, h_1, h_2, k_1, k_2), (w, x, y, z_1, z_2)).$$

4.3 Ciphertext Representation

Consider a public key $(P, q, g_1, g_2, c, d, h_1, h_2, k_1, k_2)$ for the *ACE* encryption scheme, as described in §4.1. A ciphertext of the *ACE* encryption scheme has the form

$$(s, u_1, u_2, v, e),$$

where the components are as follows:

u_1, u_2, v – integers in $\{1, \ldots, P-1\}$ (whose multiplicative order modulo P divides q).

s – an element of $\mathbf{W}^4$.

e – an element of $\mathbf{B}^*$.

We call the s, u_1, u_2, v the *preamble*, and e the *cryptogram*. If a cleartext is an l-byte string, then the length of e is $l + 16\lceil l/1024\rceil$.

We introduce the function *CEncode* that is used to map a ciphertext to its byte-string representation, and the inverse function *CDecode*. For integer $l > 0$, word string $s \in \mathbf{W}^4$, integers $0 \le u_1, u_2, v < 256^l$, and byte string $e \in \mathbf{B}^*$,

$$\begin{aligned} \mathit{CEncode}(l, s, u_1, u_2, v, e) &\stackrel{\text{def}}{=} I_{\mathbf{W}^*}^{\mathbf{B}^*}(s) \,\|\, \mathit{pad}_l(I_{\mathbf{Z}}^{\mathbf{B}^*}(u_1)) \,\|\, \mathit{pad}_l(I_{\mathbf{Z}}^{\mathbf{B}^*}(u_2)) \,\|\, \mathit{pad}_l(I_{\mathbf{Z}}^{\mathbf{B}^*}(v)) \,\|\, e \\ &\in \mathbf{B}^*. \end{aligned}$$

For integer $l > 0$ and byte string $\psi \in \mathbf{B}^*$ with $L(\psi) \ge 3l + 16$,

$$\begin{aligned} \mathit{CDecode}(l, \psi) &\stackrel{\text{def}}{=} (I_{\mathbf{B}^*}^{\mathbf{W}^*}([\psi]_0^{16}), I_{\mathbf{B}^*}^{\mathbf{Z}}([\psi]_{16}^{16+l}), I_{\mathbf{B}^*}^{\mathbf{Z}}([\psi]_{16+l}^{16+2l}), I_{\mathbf{B}^*}^{\mathbf{Z}}([\psi]_{16+2l}^{16+3l}), [\psi]_{16+3l}^{L(\psi)}) \\ &\in \mathbf{W}^4 \times \mathbf{Z} \times \mathbf{Z} \times \mathbf{Z} \times \mathbf{B}^*. \end{aligned}$$

4.4 Encryption Operation

Algorithm 4.4.1 uses an *ACE* encryption public key to encrypt a message, and outputs the resulting ciphertext.

Algorithm 4.4.1 *ACE asymmetric encryption operation.*

Input: A public key $(P, q, g_1, g_2, c, d, h_1, h_2, k_1, k_2)$ as described in §4.1, and a byte string $M \in \mathbf{B}^*$.

Output: The byte-string encoded ciphertext ψ of M as described in §4.3.

1. Generate $r \in \{0, \ldots, q-1\}$ at random.

2. Generate the ciphertext preamble:

2.1 Generate $s \in \mathbf{W}^4$ at random.

2.2 Compute $u_1 \leftarrow g_1{}^r \text{ rem } P$, $u_2 \leftarrow g_2{}^r \text{ rem } P$.

2.3 Compute $\alpha \leftarrow \mathit{UOWHash}'(k_1, L_{\mathbf{B}}(P), s, u_1, u_2) \quad \in \mathbf{Z}$ (using Algorithm 4.9.2); note that $0 \leq \alpha < 2^{160}$.

2.4 Compute $v \leftarrow c^r d^{\alpha r} \text{ rem } P$.

3. Compute the key for the symmetric encryption operation:

3.1 $\tilde{h}_1 \leftarrow h_1{}^r \text{ rem } P$, $\tilde{h}_2 \leftarrow h_2{}^r \text{ rem } P$.

3.2 Compute $k \leftarrow \mathit{ESHash}(k_2, L_{\mathbf{B}}(P), s, u_1, \tilde{h}_1, \tilde{h}_2) \in \mathbf{W}^8$ (using Algorithm 4.7.1).

4. Compute the cryptogram $e \leftarrow \mathit{SEnc}(k, s, 1024, M)$ as described in Algorithm 4.4.2.

5. Encode the ciphertext as specified in §4.3:

$$\psi \leftarrow \mathit{CEncode}(L_{\mathbf{B}}(P), s, u_1, u_2, v, e).$$

6. Return ψ.

Before presenting the details of the symmetric key encryption algorithm, we give a high-level description. An input message $M \in \mathbf{B}^*$ is broken up into blocks $M_1, \ldots, M_t$, where each block except possibly the last has $m = 1024$ bytes. Each block is encrypted using a stream cipher, yielding encrypted blocks $E_1, \ldots, E_t$, where $L(E_i) = L(M_i)$ for $1 \leq i \leq t$. Also, for each encrypted block E_i, a 16-byte message authentication code C_i is computed. The resulting cryptogram is then

$$e = E_1 \,\|\, C_1 \,\|\, \cdots \,\|\, E_t \,\|\, C_t.$$

Thus, $L(e) = L(M) + 16\lceil L(M)/m \rceil$. Note that if $L(M) = 0$, then $L(e) = 0$.

Algorithm 4.4.2 *Symmetric encryption operation SEnc.*

Input: A tuple $(k, s, m, M) \in \mathbf{W}^8 \times \mathbf{W}^4 \times \mathbf{Z} \times \mathbf{B}^*$, with $m > 0$.

Output: $e \in \mathbf{B}^l$, $l = L(M) + 16\lceil L(M)/m \rceil$.

1. If $M = \lambda_{\mathbf{B}}$, then return $\lambda_{\mathbf{B}}$.

2. Initialize a pseudo-random generator state, using Algorithm 4.6.1:

$$\mathit{genState} \leftarrow \mathit{InitGen}(k, s) \in \mathbf{GenState}.$$

3. Generate the *AXUHash* key k_{AXU} (using Algorithm 4.6.3):

$$(k_{AXU}, \mathit{genState}) \leftarrow \mathit{GenWords}((5L_{\mathbf{b}}(\lceil m/64 \rceil) + 24), \mathit{genState}).$$

4. $e \leftarrow \lambda_{\mathbf{B}}, \quad i \leftarrow 0$.

5. While $i < L(M)$ perform the following:

5.1 $r \leftarrow \min(L(M) - i, m)$.

5.2 Generate mask values for the encryption and MAC:

5.2.1 $(\mathit{mask}_m, \mathit{genState}) \leftarrow \mathit{GenWords}(4, \mathit{genState})$.

5.2.2 $(\mathit{mask}_e, \mathit{genState}) \leftarrow \mathit{GenBytes}(r, \mathit{genState})$ (using Algorithm 4.6.2).

5.3 Encrypt the plaintext: $\mathit{enc} \leftarrow [M]_i^{i+r} \oplus \mathit{mask}_e$.

5.4 Generate the message authentication code:

5.4.1 If $i + r = L(M)$, then $\mathit{lastBlock} \leftarrow 1$; otherwise $\mathit{lastBlock} \leftarrow 0$.

5.4.2 $\mathit{mac} \leftarrow \mathit{AXUHash}(k_{AXU}, \mathit{lastBlock}, \mathit{enc}) \in \mathbf{W}^4$ (using Algorithm 4.8.1).

5.5 Update the ciphertext: $e \quad \leftarrow \quad e \,\|\, \mathit{enc} \,\|\, I_{\mathbf{W}^*}^{\mathbf{B}^*}(\mathit{mac} \oplus \mathit{mask}_m)$.

5.6 $i \leftarrow i + r$.

6. Return e.

4.5 Decryption Operation

Algorithm 4.5.1 uses an ACE encryption key pair to decrypt messages that have been encrypted with the corresponding public key according to Algorithm 4.4.1.

Algorithm 4.5.1 *ACE decryption operation.*

Input: A public key $(P, q, g_1, g_2, c, d, h_1, h_2, k_1, k_2)$ and corresponding private key (w, x, y, z_1, z_2) as described in §4.1, as well as a byte string $\psi \in \mathbf{B}^*$.

Output: The decryption $M \in \mathbf{B}^* \cup \{\mathsf{Reject}\}$ of ψ.

1. Decode the ciphertext as specified in §4.3:

1.1 If $L(\psi) < 3 \cdot L_{\mathbf{B}}(P) + 16$, then return Reject.

1.2 Compute

$$(s, u_1, u_2, v, e) \leftarrow \mathit{CDecode}(L_{\mathbf{B}}(P), \psi) \quad \in \mathbf{W}^4 \times \mathbf{Z} \times \mathbf{Z} \times \mathbf{Z} \times \mathbf{B}^*;$$

note that $0 \leq u_1, u_2, v < 256^l$, where $l = L_{\mathbf{B}}(P)$.

2. Verify the ciphertext preamble:

2.1 If $u_1 \geq P$ or $u_2 \geq P$ or $v \geq P$ then return Reject.

2.2 If ${u_1}^q \neq 1 \text{ rem } P$, then return Reject.

2.3 $\mathit{reject} \leftarrow 0$.

2.4 If $u_2 \neq {u_1}^w \text{ rem } P$, then $\mathit{reject} \leftarrow 1$.

2.5 Compute $\alpha \leftarrow \mathit{UOWHash}'(k_1, L_{\mathbf{B}}(P), s, u_1, u_2) \quad \in \mathbf{Z}$ (using Algorithm 4.9.2); note that $0 \leq \alpha < 2^{160}$.

2.6 If $v \neq {u_1}^{x + \alpha y} \text{ rem } P$, then $\mathit{reject} \leftarrow 1$.

2.7 If $\mathit{reject} = 1$, then return Reject.

3. Compute the key for the symmetric decryption operation:

   3.1 $\tilde{h}_1 \leftarrow {u_1}^{z_1} \text{ rem } P,\ \tilde{h}_2 \leftarrow {u_1}^{z_2} \text{ rem } P$.

   3.2 Compute $k \leftarrow \mathit{ESHash}(k_2, L_{\mathbf{B}}(P), s, u_1, \tilde{h}_1, \tilde{h}_2) \in \mathbf{W}^8$ (using Algorithm 4.7.1).

4. Compute $M \leftarrow \mathit{SDec}(k, s, 1024, e)$ as described in Algorithm 4.5.2; note that *SDec* may return Reject.

5. Return M.

Algorithm 4.5.2 *Decryption operation SDec.*

Input: A tuple $(k, s, m, e) \in \mathbf{W}^8 \times \mathbf{W}^4 \times \mathbf{Z} \times \mathbf{B}^*$, with $m > 0$.

Output: The decryption $M \in \mathbf{B}^* \cup \{\text{Reject}\}$ of e.

1. If $e = \lambda_{\mathbf{B}}$, then return $\lambda_{\mathbf{B}}$.

2. Initialize a pseudo-random generator state, using Algorithm 4.6.1:

$$\mathit{genState} \leftarrow \mathit{InitGen}(k, s) \in \mathbf{GenState}.$$

3. Generate the *AXUHash* key k_{AXU} (using Algorithm 4.6.3):

$$(k_{AXU}, \mathit{genState}') \leftarrow \mathit{GenWords}((5L_{\mathbf{b}}(\lceil m/64 \rceil) + 24), \mathit{genState}).$$

4. $M \leftarrow \lambda_{\mathbf{B}}, \quad i \leftarrow 0$.

5. While $i < L(e)$ perform the following:

   5.1 $r \leftarrow \min(L(e) - i, m + 16) - 16$.

   5.2 If $r \leq 0$, then return Reject.

   5.3 Generate mask values for the encryption and MAC:

      5.3.1 $(\mathit{mask}_m, \mathit{genState}) \leftarrow \mathit{GenWords}(4, \mathit{genState})$.

      5.3.2 $(\mathit{mask}_e, \mathit{genState}) \leftarrow \mathit{GenBytes}(r, \mathit{genState})$ (using Algorithm 4.6.2).

   5.4 Verify the message authentication code:

      5.4.1 If $i + r + 16 = L(M)$, then $\mathit{lastBlock} \leftarrow 1$; otherwise $\mathit{lastBlock} \leftarrow 0$.

      5.4.2 $\mathit{mac} \leftarrow \mathit{AXUHash}(k_{AXU}, \mathit{lastBlock}, [e]_i^{i+r}) \in \mathbf{W}^4$ (using Algorithm 4.8.1).

      5.4.3 If $[e]_{i+r}^{i+r+16} \neq I_{\mathbf{W}^*}^{\mathbf{B}^*}(\mathit{mac} \oplus \mathit{mask}_m)$, then return Reject.

   5.5 Update the plaintext: $M \leftarrow M \,\|\, ([e]_i^{i+r} \oplus \mathit{mask}_e)$.

   5.6 $i \leftarrow i + r + 16$.

6. Return M.

4.6 Pseudo-Random Bit Generator

This section defines a pseudo-random bit generator, based on the block cipher *MARS*. The state of the generator is an element of the set

$$\mathbf{GenState} = \mathbf{W}^8 \times \mathbf{W}^4 \times \mathbf{B}^{16} \times \{0, \ldots, 16\}.$$

It produces an unlimited sequence of bytes. The generator works by using *MARS* in "sum/counter mode," but with a randomized starting value.

First comes the initialization routine. The first input parameter k should be random and secret—it is used as a *MARS* key. The second input parameter s should be random, but need not be secret—it is used to initialize a counter.

Algorithm 4.6.1 *Pseudo-Random Bit Generator: InitGen.*

Input: A tuple $(k, s) \in \mathbf{W}^8 \times \mathbf{W}^4$.

Output: A state $genState \in \mathbf{GenState}$.

1. $genState \leftarrow (k, s, 0_{\mathbf{B}^{16}}, 16) \in \mathbf{GenState}$.
2. Return $genState$.

The next algorithm is used to generate pseudo-random byte strings.

Algorithm 4.6.2 *Pseudo-Random Bit Generator: GenBytes.*

Input: $(n, genState) \in \mathbf{Z} \times \mathbf{GenState}$, with $n \geq 0$.

Output: $(out_b, genState')$, where $out_b \in \mathbf{B}^n$ and $genState' \in \mathbf{GenState}$ is the new state of the generator.

1. Set
$$(k, s, buf, iread) \leftarrow genState \ \in \mathbf{W}^8 \times \mathbf{W}^4 \times \mathbf{B}^{16} \times \{0, \ldots, 16\}.$$
2. Set $out_b \leftarrow \lambda_{\mathbf{B}}$.
3. While $n > 0$ do the following:

   3.1 If $iread \geq 16$, re-load the buffer:

   3.1.1 $buf \leftarrow I_{\mathbf{W}^*}^{\mathbf{B}^*}(MARS(k, s))$.

   3.1.2 $s \leftarrow s + 1$.

   3.1.3 $buf \leftarrow buf \oplus I_{\mathbf{W}^*}^{\mathbf{B}^*}(MARS(k, s))$.

   3.1.4 $s \leftarrow s + 1$.

   3.1.5 $iread \leftarrow 0$.

   3.2 Accumulate up to 16 output bytes:

   3.2.1 $r \leftarrow \min(iread + n, 16)$.

   3.2.2 $out_b \leftarrow out_b \, \| \, [buf]_{iread}^{r}$.

3.2.3 $n \leftarrow n - r + iread, \quad iread \leftarrow r.$

4. $genState' \leftarrow (k, s, buf, iread)$.
5. Return $(out_b, genState')$.

For convenience, the following variation outputs word strings.

Algorithm 4.6.3 *Pseudo-Random Bit Generator: GenWords.*

Input: $(n, genState) \in \mathbf{Z} \times \mathbf{GenState}$, with $n \geq 0$.

Output: $(out_w, genState')$, where $out_w \in \mathbf{W}^n$ and $genState' \in \mathbf{GenState}$ is the new state of the generator.

1. Compute $(out_b, genState') \leftarrow GenBytes(4n, genState)$ using Algorithm 4.6.2.
2. Set $out_w \leftarrow I_{\mathbf{B}^*}^{\mathbf{W}^*}(out_b)$.
3. Return $(out_w, genState')$.

4.7 Entropy-Smoothing Hash Function

This section defines an entropy-smoothing hash function.

Algorithm 4.7.1 *Entropy smoothing hash transformation ESHash.*

Input: A tuple $(k, l, s, u_1, \tilde{h}_1, \tilde{h}_2) \in \mathbf{B}^* \times \mathbf{Z} \times \mathbf{W}^4 \times \mathbf{Z} \times \mathbf{Z} \times \mathbf{Z}$, where $L(k) = 32m + 40$ for some integer m with $m \geq \lceil l/16 \rceil$, and $0 \leq \tilde{h}_1, \tilde{h}_2, u_1 < 256^l$.

Output: A hash value $h \in \mathbf{W}^8$.

1. Set $l_1 \leftarrow \lceil l/4 \rceil, \quad l_2 \leftarrow \lceil l_1/4 \rceil, \quad l_3 \leftarrow \lceil (3l_1 + 4)/16 \rceil$.
2. $k' \leftarrow I_{\mathbf{B}^*}^{\mathbf{W}^*}(k)$.
3. Encode $(s, u_1, \tilde{h}_1, \tilde{h}_2)$ as a word string M, padding to a multiple of 16 words:
$$M \leftarrow pad_{16l_3}\left(s \,\|\, pad_{l_1}(I_{\mathbf{Z}}^{\mathbf{W}^*}(u_1)) \,\|\, pad_{l_1}(I_{\mathbf{Z}}^{\mathbf{W}^*}(\tilde{h}_1)) \,\|\, pad_{l_1}(I_{\mathbf{Z}}^{\mathbf{W}^*}(\tilde{h}_2))\right) \in \mathbf{W}^{16l_3}.$$
4. Compute a simplified SHA-1 hash (twice):

   4.1 $s \leftarrow [k']_0^5$.

   4.2 For $i = 1$ to l_3 do: $s \leftarrow CSHA1(s, [M]_{16(i-1)}^{16i})$.

   4.3 $s' \leftarrow [k']_5^{10}$.

   4.4 For $i = 1$ to l_3 do: $s' \leftarrow CSHA1(s', [M]_{16(i-1)}^{16i})$.
5. Encode $(\tilde{h}_1, \tilde{h}_2)$ as a word string M', padding to a multiple of 8 words:
$$M' \leftarrow pad_{8l_2}\left(pad_{l_1}(I_{\mathbf{Z}}^{\mathbf{W}^*}(\tilde{h}_1)) \,\|\, pad_{l_1}(I_{\mathbf{Z}}^{\mathbf{W}^*}(\tilde{h}_2))\right) \in \mathbf{W}^{8l_2}.$$

6. Compute

$$c \leftarrow \sum_{i=1}^{l_2} \left(I_{\mathbf{W}^*}^{\mathbf{F}_2[T]}([M']_{8(i-1)}^{8i}) I_{\mathbf{W}^*}^{\mathbf{F}_2[T]}([k']_{8i+2}^{8i+10}) \right) \text{ rem } f \quad \in \mathbf{F}_2[T],$$

where $f = T^{256} + T^{10} + T^5 + T^2 + 1$.

7. Compute $h \leftarrow pad_8(I_{\mathbf{F}_2[T]}^{\mathbf{W}^*}(c)) \oplus (s \,\|\, [s']_0^3) \;\in \mathbf{W}^8$.

8. Return h.

4.8 AXU Hash Function

This section defines an "almost XOR-universal hash function," denoted *AXUHash*.

Algorithm 4.8.1 *Almost XOR-universal hash function AXUHash.*

Input: A tuple $(k, lastBlock, M) \in \mathbf{W}^* \times \{0,1\} \times \mathbf{B}^*$, where $L(M) > 0$, and $L(k) = 5m + 24$ for some integer $m \geq L_{\mathbf{b}}(\lceil L(M)/64 \rceil)$.

Output: The hash value $res \in \mathbf{W}^4$ of M under the key k.

1. Compute $h \leftarrow UOWHash([k]_0^{L(k)-8}, I_{\mathbf{B}^*}^{\mathbf{W}^*}(pad_l(M))) \in \mathbf{W}^5$, where $l = 64\lceil L(M)/64 \rceil$, using Algorithm 4.9.1.
2. $c_1 \leftarrow I_{\mathbf{W}^*}^{\mathbf{F}_2[T]}([h]_0^4) \quad \in \mathbf{F}_2[T]$.
3. $d_1 \leftarrow I_{\mathbf{W}^*}^{\mathbf{F}_2[T]}([k]_{L(k)-8}^{L(k)-4}) \quad \in \mathbf{F}_2[T]$.
4. $c_2 \leftarrow I_{\mathbf{W}^*}^{\mathbf{F}_2[T]}([h]_4^5 \,\|\, I_{\mathbf{Z}}^{\mathbf{W}^*}(2 \cdot L(M) + lastBlock)) \quad \in \mathbf{F}_2[T]$.
5. $d_2 \leftarrow I_{\mathbf{W}^*}^{\mathbf{F}_2[T]}([k]_{L(k)-4}^{L(k)}) \quad \in \mathbf{F}_2[T]$.
6. $res \leftarrow pad_4(I_{\mathbf{F}_2[T]}^{\mathbf{W}^*}((c_1 d_1 + c_2 d_2) \text{ rem } f))$, where $f = T^{128} + T^7 + T^2 + T + 1$.
7. Return res.

4.9 Universal One-Way Hash Function

This section defines a universal one-way hash function.

First comes a "low level" version, denoted *UOWHash*, that performs no length encoding or padding on the message input.

Algorithm 4.9.1 *Universal one-way hash function UOWHash.*

Input: A tuple $(k, M) \in \mathbf{W}^* \times \mathbf{W}^*$, where $L(M) = 16n$ for some integer $n > 0$, and $L(k) = 5m + 16$ for some integer $m \geq L_{\mathbf{b}}(n)$.

Output: The hash value $h \in \mathbf{W}^5$ of M under key k.

1. Initialize $h \leftarrow 0_{\mathbf{W}^5} \in \mathbf{W}^5$, $\;msk \leftarrow [k]_0^{16} \in \mathbf{W}^{16}$.

2. For $i = 1$ to n do the following:

   2.1 Compute the key index j such that $i = 2^j d$ for odd $d \in \mathbf{Z}$.

   2.2 Compute the next initial SHA-1 hash state $s \leftarrow h \oplus [k]_{5j+16}^{5j+21} \in \mathbf{W}^5$.

   2.3 Compute a SHA-1 input block $m \leftarrow [M]_{16(i-1)}^{16i} \oplus msk \quad \in \mathbf{W}^{16}$.

   2.4 Perform the core SHA-1 state transformation: $h \leftarrow \mathit{CSHA1}(s, m)$.

3. Return h.

Next comes $\mathit{UOWHash}'$ which encodes its input in a special way before calling $\mathit{UOWHash}$.

Algorithm 4.9.2 *Universal one-way hash function* $\mathit{UOWHash}'$.

Input: A tuple $(k, l, s, u_1, u_2) \in \mathbf{B}^* \times \mathbf{Z} \times \mathbf{W}^4 \times \mathbf{Z} \times \mathbf{Z}$, where $l > 0$, $0 \leq u_1, u_2 < 256^l$, $L(k) = 20L_{\mathbf{b}}(\lceil(2\lceil l/4\rceil + 4)/16\rceil) + 64$.

Output: The hash value $a \in \mathbf{Z}$, where $0 \leq a < 2^{160}$.

1. Set $l_1 \leftarrow \lceil l/4\rceil, \quad l_2 \leftarrow \lceil(2\lceil l/4\rceil + 4)/16\rceil$.

2. Encode (s, u_1, u_2) as a word string, padding to a multiple of 16 words:

$$u \leftarrow \mathit{pad}_{16l_2}\left(s \,\|\, \mathit{pad}_{l_1}(I_{\mathbf{Z}}^{\mathbf{W}^*}(u_1)) \,\|\, \mathit{pad}_{l_1}(I_{\mathbf{Z}}^{\mathbf{W}^*}(u_2))\right) \quad \in \mathbf{W}^{16l_2}.$$

3. Compute

$$a' \leftarrow \mathit{UOWHash}(I_{\mathbf{B}^*}^{\mathbf{W}^*}(k), u) \quad \in \mathbf{W}^5,$$

   using Algorithm 4.9.1.

4. Compute $a \leftarrow I_{\mathbf{W}^*}^{\mathbf{Z}}(a') \in \mathbf{Z}$.

5. Return a.

4.10 Security analysis

We analyze the security properties of the above encryption scheme.

The concrete security of our encryption scheme is straightforward, if somewhat tedious, to analyze, based upon the arguments in [CS98] and [Sho00a]. Consider an adversary that runs in time at most t, makes at most κ decryption requests, and presents test messages whose length in bytes is at most l. The adversary's advantage, $\mathsf{AdvEnc}(t, \kappa, l)$ (as defined in §2.2) can be explicitly bounded in terms of

- the advantage the adversary has in solving the DDH (see AdvDDH, defined in §2.5),
- the advantage the adversary has in finding second preimages in SHA-1 (see AdvSHA, defined in §2.7), and

- the advantage the adversary has in distinguishing MARS output from random (see $\mathsf{AdvMARS}$, defined in 2.8).

Also, we let $l' = L_{\mathbf{B}}(P)$. Recall that q is the order of the subgroup of the multiplicative group of units modulo P in which we are working.

Theorem 4.10.1 *We have:*

$$\begin{aligned}\mathsf{AdvEnc}(t, \kappa, l) \quad \leq \quad & \mathsf{AdvDDH}(O(t)) + \\ & \mathsf{AdvSHA}(O(t))(\lceil l/64 \rceil + \lceil (2\lceil l'/4 \rceil + 4)/16 \rceil) + \\ & \mathsf{AdvMARS}(O(t), 65\lceil l/1024 \rceil + 7) \cdot 2 + \\ & \frac{2\kappa + 1}{q} + \\ & \frac{\kappa + 2}{2^{128}}. \end{aligned} \tag{1}$$

The running times $O(t)$ reflect the running times of simulators that do little more than run the adversary, plus just a little additional bookkeeping which can effectively be ignored.

We shall prove this theorem, referring the reader at times to arguments in [CS98] and [Sho00a]. We can assume $l > 0$, since otherwise the adversary's advantage is by definition zero.

We shall repeatedly make use of the following simple lemma, which we record here for convenience.

Lemma 4.10.1 *Let E, E', F, and F' be events defined on a probability space such that* $\Pr[E|\neg F] = \Pr[E'|\neg F']$ *and* $\epsilon = \Pr[F] = \Pr[F']$. *Then we have*

$$\Big| \Pr[E] - \Pr[E'] \Big| \leq \epsilon.$$

This follows from a simple calculation. We have

$$\Pr[E] = \Pr[E|\neg F](1 - \epsilon) + \Pr[E|F]\epsilon$$

and

$$\Pr[E'] = \Pr[E'|\neg F'](1 - \epsilon) + \Pr[E'|F']\epsilon.$$

Subtracting these two equations and taking absolute values, we have

$$\Big| \Pr[E] - \Pr[E'] \Big| = \epsilon \Big| \Pr[E|F] - \Pr[E'|F'] \Big| \leq \epsilon.$$

That completes the proof of the lemma.

Some notational conventions. Recall that a ciphertext ψ is of the form $\psi = (s, u_1, u_2, v, e)$, as described in §4.3. Recall also that $\pi = (s, u_1, u_2, v)$ is called the *preamble* of ψ, and e is called the *cryptogram* of ψ. In the proof below, whenever we refer to a generic ciphertext ψ, the values s, u_1, u_2, v, e, as well as π, are implicitly defined as above. Also implicitly defined is the hash value α of (s, u_1, u_2), as computed in step 2 of Algorithm 4.5.1, as well as the values $\tilde{h}_1$, $\tilde{h}_2$, and k, as computed in step 3 of Algorithm 4.5.1. We shall always refer to the *target ciphertext*, i.e., the ciphertext output by the encryption oracle in the attack, as ψ', and the values

$$s', u_1', u_2', v', e', \pi', \alpha', \tilde{h}_1', \tilde{h}_2', k'$$

are analogously defined for the target ciphertext.

The following definition is also convenient.

Definition 4.10.1 *A ciphertext $\psi = (s, u_1, u_2, v, e)$ is called* **valid** *if $\log_{g_1} u_1 = \log_{g_2} u_2$, where the discrete logarithms are with respect to the multiplicative group of units modulo P; otherwise, ψ is* **invalid**.

We now turn to the proof of the theorem.

Consider the attack game defined in §2.2 with respect to a specific adversary that runs in time at most t, makes at most κ decryption requests, and submits test messages of length at most l.

Call the original attack game G_0. Let S_0 be the event that the adversary guesses the value of the hidden bit b in game G_0. We have

$$\mathsf{AdvEnc}(t, \kappa, l) = \left| \Pr[S_0] - 1/2 \right|. \tag{2}$$

We shall make several transformations of the game, obtaining games G_1, G_2, etc. In order to relate probabilities of certain events in different games, conceptually, these games all are run on the same underlying probability distribution—only the computation rules change. In each game G_i, for $i = 1, 2$, etc., we let S_i denote the event that the adversary guesses the value of the hidden bit b in game G_i.

Game G_1. In the first transformation, game G_1, we replace the the private key by

$$x_1, x_2, y_1, y_2, z_{11}, z_{12}, z_{21}, z_{22},$$

where each of these is chosen at random modulo q. Also, we compute the public key as follows. We choose g_1, g_2 to be random numbers whose order modulo P is equal to q. Then we compute

$$c \leftarrow g_1^{x_1} g_2^{x_2} \text{ rem } P,\ d \leftarrow g_1^{y_1} g_2^{y_2} \text{ rem } P,\ h_1 \leftarrow g_1^{z_{11}} g_2^{z_{12}} \text{ rem } P,\ h_2 \leftarrow g_1^{z_{21}} g_2^{z_{22}} \text{ rem } P.$$

Further, in the decryption algorithm, we verify the ciphertext preamble (step 2 in Algorithm 4.5.1) with the following test:

$$u_1^q \equiv 1 \pmod{P},\ u_2^q \equiv 1 \pmod{P},\ \text{and } u_1^{x_1 + y_1 \alpha} u_2^{x_2 + y_2 \alpha} \equiv v \pmod{P}.$$

Finally, in the derivation of the decryption key (step 3.2 in Algorithm 4.4.1 and step 3.2 in Algorithm 4.5.1), we compute

$$\tilde{h}_1 \leftarrow u_1^{z_{11}} u_2^{z_{12}} \text{ rem } P,\ \tilde{h}_2 \leftarrow u_1^{z_{21}} u_2^{z_{22}} \text{ rem } P.$$

That completes the description of game G_1. We view G_1 and G_0 as operating on a common probability space defined in terms of the variables

$$w, x, y, z_1, z_2,$$

and

$$x_1, x_2, y_1, y_2, z_{11}, z_{12}, z_{21}, z_{22},$$

where the first set of variables are only implicitly defined in G_1 and the second set of variables are only implicitly defined in G_0. Let U_1 to be event that some invalid ciphertext is not rejected in game G_1. Following the arguments in [CS98], the probability that any single invalid ciphertext is not rejected is at most $1/q$, from which it follows that

$$\Pr[U_1] \le \frac{\kappa}{q}. \tag{3}$$

Also, one can easily check that so long as event U_1 does not occur, the adversary's attack in game G_1 proceeds just as in game G_0. That is,

$$\Pr[S_1 | \neg U_1] = \Pr[S_0 | \neg U_1]. \tag{4}$$

Now apply Lemma 4.10.1 with $(E, E', F, F') = (S_0, S_1, U_1, U_1)$, and we obtain

$$\Big| \Pr[S_1] - \Pr[S_0] \Big| \le \frac{\kappa}{q}. \tag{5}$$

Game G_2. In the second transformation, game G_2, we modify the behavior of the encryption oracle in the same way as is done in the security argument in [CS98]. That is, in computing ψ', instead of following the encryption algorithm, we simply choose u_1' and u_2' as random numbers whose order modulo P divides q. Also, the encryption oracle computes v' using the algorithm used by the decryption algorithm:

$$v' \leftarrow (u_1')^{x_1 + y_1 \alpha'} (u_2')^{x_2 + y_2 \alpha'} \text{ rem } P.$$

As in [CS98], one easily verifies that

$$\Big| \Pr[S_2] - \Pr[S_1] \Big| \le \mathsf{AdvDDH}(O(t)). \tag{6}$$

Game G_3. In the third transformation, game G_3, we modify game G_2 as follows. Let V_2 be the event that that the adversary in game G_2 ever submits a ciphertext ψ for decryption with $(s, u_1, u_2) \neq (s', u_1', u_2')$, but with $\alpha = \alpha'$. In game G_3, we move the computation of π' (along with the derived values α', $\tilde{h}_1'$, $\tilde{h}_2'$, and k') to the very beginning of the attack, and if event V_2 occurs, we simply stop the attack. From the analysis in [Sho00a], we have

$$\Pr[V_2] \le \mathsf{AdvSHA}(O(t)) \cdot \lceil (2\lceil l'/4 \rceil + 4)/16 \rceil. \tag{7}$$

Note that the quantity $\lceil (2\lceil l'/4 \rceil + 4)/16 \rceil$ is the number of 512-bit input blocks to the hash function. Because of the way G_3 was derived from G_2, one easily verifies that

$$\Pr[S_2 | \neg V_2] = \Pr[S_3 | \neg V_2]. \tag{8}$$

Applying Lemma 4.10.1 with $(E, E', F, F') = (S_2, S_3, V_2, V_2)$, we obtain

$$\Big| \Pr[S_3] - \Pr[S_2] \Big| \le \mathsf{AdvSHA}(O(t)) \cdot \lceil (2\lceil l'/4 \rceil + 4)/16 \rceil. \tag{9}$$

Game G_4. In the next transformation, game G_4, we modify the encryption oracle yet again. Instead of computing $\tilde{h}_1'$ and $\tilde{h}_2'$ as in the encryption algorithm, we simply choose them as random numbers whose order modulo P divides q. Let W_3 be the event that either

- $\log_{g_1} u_1' = \log_{g_2} u_2'$ in game G_3, or
- some invalid ciphertext ψ with $\pi \neq \pi'$ is not rejected in game G_3.

Note that the target ciphertext ψ' is itself invalid when $\log_{g_1} u_1' \neq \log_{g_2} u_2'$. From the analysis in [CS98], the probability that any single invalid ciphertext is not rejected, given that $\log_{g_1} u_1' \neq \log_{g_2} u_2'$, is at most $1/q$, from which it follows that

$$\Pr[W_3] \leq \frac{\kappa + 1}{q}. \tag{10}$$

We can define an analogous event W_4 for game G_4. Note that events W_3 and W_4 are not the same; nevertheless, by the analysis in [CS98], one sees that

$$\Pr[W_3] = \Pr[W_4] \text{ and } \Pr[S_4|\neg W_4] = \Pr[S_3|\neg W_3]. \tag{11}$$

Applying Lemma 4.10.1 with $(E, E', F, F') = (S_3, S_4, W_3, W_4)$, we obtain

$$\Big|\Pr[S_4] - \Pr[S_3]\Big| \leq \frac{\kappa + 1}{q}. \tag{12}$$

Game G_5. In the next transformation, game G_5, we replace the derived symmetric key k' computed by the encryption oracle by a random key. Also, when the decryption oracle is presented with a ciphertext ψ with $\pi = \pi'$, it uses the same random key k'. By the Entropy Smoothing Theorem (a.k.a., the Leftover Hash Lemma; see Chapter 8 of [Lub96] or [IZ89]), and the fact that $(\tilde{h}_1', \tilde{h}_2')$ is chosen at random from a set of size at least 2^a, where $a = 2 \times 255 = 256 + 2 \times 127$, we have

$$\Big|\Pr[S_5] - \Pr[S_4]\Big| \leq \frac{2}{2^{128}}. \tag{13}$$

Game G_6. In the next transformation, game G_6, we modify the decryption oracle as follows. Suppose the decryption oracle is presented with a ciphertext ψ with $\pi = \pi'$ and $L(e) \neq 0$. Then we simply let the decryption oracle reject ψ. Let X_5 be the event that such a ciphertext ψ is not rejected in game G_5. We claim that

$$\begin{aligned}\Pr[X_5] \quad \leq \quad & \mathsf{AdvMARS}(O(t), 65\lceil l/1024\rceil + 7) + \\ & \mathsf{AdvSHA}(O(t)) \cdot \lceil l/64\rceil + \\ & \frac{\kappa}{2^{128}}.\end{aligned} \tag{14}$$

From this, it will follow by an application of Lemma 4.10.1 with $(E, E', F, F') = (S_5, S_6, X_5, X_5)$ that

$$\Big|\Pr[S_6] - \Pr[S_5]\Big| \leq \Pr[X_5]. \tag{15}$$

To prove (14), first recall that a cryptogram is split into 1024-byte blocks, and each block is individually authenticated using a message authentication code (MAC). Also note that not only is the content of each block authenticated, but also its status as the last block, and its length (which is only relevant in case the block is the last block of the message). Suppose the target cryptogram consists of b blocks, i.e., $b = \lceil l/1024\rceil$. Let Y be the event that for some ψ submitted for decryption, with $L(e) \neq 0$ and $\pi = \pi'$, either

- ψ' has not yet been generated and the first block of e has a valid MAC, or
- ψ' has been generated, and the first block of e that differs from that of e' has a valid MAC.

We observe that

$$\Pr[X_5] \leq \Pr[Y]. \tag{16}$$

To bound Y, we make a transformational argument, defining a sequence of transformed games $G_5^{(1)}$, $G_5^{(2)}$, $G_5^{(3)}$, and defining the events $Y^{(i)}$, for $i = 1, 2, 3$, to be the events corresponding to Y, but in game $G_5^{(i)}$. First, we replace G_5 by the game $G_5^{(1)}$ in which we halt the game as soon as event Y occurs. Clearly,

$$\Pr[Y^{(1)}] = \Pr[Y]. \tag{17}$$

Note that in game $G_5^{(1)}$, when the decryption oracle is presented with a ciphertext ψ with $\pi = \pi'$, it never processes more than b blocks of the cryptogram e. Second, we replace game $G_5^{(1)}$ with game $G_5^{(2)}$, in which the output of the pseudo-random bit generator in the encryption oracle is first extended (by less than 1024 bytes) so as to cover b full blocks of text, and is then replaced by a random string of the same length. The same random bit string is used by the decryption oracle whenever a ciphertext ψ with $\pi = \pi'$ is presented for decryption. We have

$$\left| \Pr[Y^{(2)}] - \Pr[Y^{(1)}] \right| \leq \mathsf{AdvMARS}(O(t), 65\lceil l/1024 \rceil + 7). \tag{18}$$

Next, $G_5^{(2)}$ is replaced by the game $G_5^{(3)}$ in which the adversary is modified so as to simply halt if ψ' has already been generated, and the evaluation of $AXUHash$ during decryption of a ciphertext ψ with $\pi = \pi'$ and $L(e) = L(e')$ produces a collision in SHA-1. Again using the analysis in [Sho00a], an application of Lemma 4.10.1 yields

$$\left| \Pr[Y^{(3)}] - \Pr[Y^{(2)}] \right| \leq \mathsf{AdvSHA}(O(t)) \cdot \lceil l/64 \rceil. \tag{19}$$

Finally, using standard arguments for message authentication codes based on universal hashing (see, e.g., [Kra94]), one sees that

$$\Pr[Y^{(3)}] \leq \frac{\kappa}{2^{128}}. \tag{20}$$

Inequality (14) now follows directly from in inequalities (16), (17), (18), (19), and (20).

Game G_7. In the final transformation, game G_7, we simply modify game G_6 so that the output of the pseudo-random bit generator in the encryption oracle is replaced by a random string of corresponding length. Then we have

$$\left| \Pr[S_7] - \Pr[S_6] \right| \leq \mathsf{AdvMARS}(O(t), 65\lceil l/1024 \rceil + 7). \tag{21}$$

It is easy to see that

$$\Pr[S_7] = \frac{1}{2}. \tag{22}$$

The theorem now follows from inequalities (2), (5), (6), (9), (12), (13), (14), (15), (21), and (22).

That completes the proof of Theorem 4.10.1

Remarks. One should note that this reduction is quite tight.

In the above calculation, we have assumed the the random numbers used by the key generation and encryption algorithms are perfect. If instead, a source of pseudo-random bits is used, then to the above advantage for breaking the encryption scheme, one must add the adversary's advantage in distinguishing these pseudo-random bits from truly random bits.

One strange thing about this theorem is the coefficient of 2 that appears in the AdvMARS term. It is not clear if this "2" cannot be replaced by a "1"; however, at the moment, we do not see how to do this.

4.11 Further discussion and implementation notes

Random oracles

As we have already mentioned in §2.4, in the random oracle model, one can replace the DDH assumption by the potentially weaker CDH assumption. The security analysis in this case can be found in [Sho00b]. We do not carry out a concrete security analysis in this case, but we note that the reduction in this case is not very efficient. But since the random oracle model is anyway a heuristic, we do not view this as a major problem.

Hiding the length of a message

Note that the encryption algorithm does not make any attempt to hide the length of a message, and indeed, the length of the cleartext is easily calculated from the length of the corresponding ciphertext. Thus an encryption of `"yes"` can easily be distinguished from an encryption of `"no"`. This problem is easily avoided by appropriately padding the cleartext (e.g., encrypting `"no␣"` instead of `"no"`). We emphasize that *it is up to the application using the encryption scheme to format and pad cleartexts as necessary so as to hide information that could be derived from the length of a message.*

Optimizations

All five of the exponentiations performed in the decryption algorithm are to the base u_1, and hence standard algorithmic techniques can be used to compute this faster than five exponentiations. Also note that in step 2.4 of Algorithm 4.4.1, the quantity $c^r d^{ar} \text{ rem } P$ can be computed faster than two exponentiations, also using standard algorithmic techniques. We refer the reader to §14.6 of [MvOV97] for these algorithmic details.

Timing information

Note that in step 2.2 in algorithm Algorithm 4.5.1, we set *reject* to 1, and delay returning from the function until later. We do this to prevent timing information from being leaked to an adversary playing in game G_0 that is not available in game G_1 (see the proof of Theorem 4.10.1). We recommend that all implementations follow a similar practice. The point of making this transformation is to get a simpler and more efficient decryption algorithm. Although this implementation prevents an adversary from potentially taking advantage of some "crude" timing information, we make absolutely no claims about its security against timing attacks [Koc96] or power analysis [KJJ99] in general.

Early detection of a corrupted ciphertext

Note that when encrypting the actual payload, we use a symmetric cipher with an authentication code. The cryptogram is broken up into 1024-byte blocks, and each of these is individually authenticated. This is done so that a receiver can stop processing a corrupted stream of encrypted data almost as soon as the corruption as occured. This seems desirable from a security point of view to the alternative approach of authenticating the message as a whole, for the following reason. While decrypting a very long message, the receiver may have to store the cleartext on disk, perhaps only to reject it. However, while the cleartext is on disk, it may be more vulnerable than it would be in main memory. Thus, it seems desirable to detect and reject a corrupted message as soon as is practicable.

Note that no useful timing information is leaked to the adversary when the processing of a corrupted stream is terminated. Intuitively, the adversary already "knows" where the stream is corrupted.

"Salted" MARS

Note that the pseudo-random bit string is derived using MARS in sum/counter mode, starting with the counter initialized to a random value s. The value s is chosen at random with every encryption. This "salting" technique should have the effect in practice of forcing any cryptanalysis on MARS to focus its efforts on individual ciphertexts. Note that to make the proof of security in the random oracle model in [Sho00b] work, it is essential that s be an input to the cryptographic hash in the entropy smoothing hash function.

The multi-user/multi-message environment

As already mentioned, at least in an asymptotic sense, the definition of security we have used implies security in a multi-user/multi-message environment. Using a standard "hybrid" argument, one sees that security essentially degrades by a factor of

$$\#\text{ number of users } \times \text{ max } \#\text{ of messages per user.} \tag{23}$$

We believe that our choices of parameters allow sufficient "head room" so that one still obtains a meaninful level of security even considering fairly large systems of users.

Our algorithm design could be somewhat improved in this regard, however. By following the suggestion in [BBM00] that all users work with a common group, and also by having all users work with the same UOWH key, one gets a quantitatively better security proof in the multi-user setting, where the security degrades by a factor proportional to the total number of messages encrypted, which may be significantly less than (23). However, this comes at a cost: all users must use the same defining paramaters, which may be both inconvenient, and also introduces a new "trust" problem. Moreover, it allows an attacker to focus all of his computational resources on a single group, which can potentially lead to a catastrophic security lapse.

Encrypting the empty message

We comment about encrypting the empty message. From a security point of view, it hardly makes sense to encrypt the empty message. Nevertheless, we allow this, if only for the sake of a flexible interface. The encryption (s, u_1, u_2, v, e) of the empty message consists of an ordinary preamble (s, u_1, u_2, v), but an empty cryptogram $e = \lambda_{\mathbf{B}}$. Note that a user may create an encryption of (s, u_1, u_2, v, e) of a non-empty message, so $e \neq \lambda_{\mathbf{B}}$, and if an adversary then submits $(s, u_1, u_2, v, \lambda_{\mathbf{B}})$ for decryption, the decryption algorithm will *accept* this ciphertext, and generate the empty message as its decryption. This behavior may seem a bit unusual, but still satisfies the definition of security.

Implementing the key generation algorithm

In the key generation algorithm, we have to generate a random prime q, and a random prime P such that $P \equiv 1 \pmod{q}$. To generate q, one can generate random numbers and apply an iterated Miller-Rabin test. To get a small error probability, one must iterate the Miller-Rabin test sufficiently many times. For this purpose, one can use the results in [DLP93].

Once q has been generated, we can iteratively choose P at random of the desired length, subject to $P \equiv 1 \pmod{q}$, and apply an iterated Miller-Rabin test to P. Note that the results in [DLP93] are not directly applicable, since P is *not* a random number of prescribed length. Instead, to obtain a k-bit prime P congruent to 1 mod q, with an error bound of ϵ, one should iterate the Miller-Rabin test t times, where $4^{-t}k/2 \leq \epsilon$. Although P is not random, since P is quite large, and $P > q^3$, one can show under the Generalized Riemann Hypothesis that the probability that a random P congruent to 1 mod q is prime is extremely close to the probability that a random number of the same length is prime (see Theorem 8.1.18 in [BS96]), and this is bounded from below by $2/k$ for all k under consideration (see the estimate, e.g., in the proof of Proposition 2 in [DLP93]). From these considerations, and the basic properties of the Miller-Rabin test, it follows that the overall error probability will be at most ϵ.

This approach is a bit crude, and unfortunately, leads to a somewhat slow key generation algorithm. It would be nice if the results of [DLP93] could be generalized to primes in arithmetic progressions, but we are unaware of any such results.

A reasonable choice of ϵ is $\epsilon = 2^{-80}$.

API considerations

We have designed the encryption and decryption algorithms so that they can work with *streams* of data. The message to be encrypted can be presented to the encryption algorithm as a stream, and the ciphertext can be generated as a stream. This ciphertext stream can be fed directly in to the decryption algorithm, which produces the cleartext as a stream.

Actually, if one employs such a streaming implementation, one must consider the possibility that the adversary might adaptively choose the latter bits of m_0, m_1 after having seen a prefix of the target ciphertext, also possibly interacting with the decryption oracle in the meantime. Our proof of security does not deal with this scenario: it assumes the adversary submits m_0, m_1 in their entirety before any prefix of the target ciphertext is obtained. However, the proof of security can be adapted to this somewhat richer attack scenario—we leave the details to the interested reader.

5 Signature Scheme

In this section, we describe the signature scheme, which is a variant of that in [CS99].

5.1 Signature Key Pair

The signature scheme defined in this document employs two key types, whose representation consists of the following tuples:

ACE Signature public key: (N, h, x, e', k', s).
ACE Signature private key: (p, q, a).

For a given size parameter m, with $1024 \leq m \leq 16,384$, the components are as follows:

p – $\lfloor m/2 \rfloor$-bit prime number with $(p-1)/2$ is also prime.

q – $\lceil m/2 \rceil$-bit prime number with $(q-1)/2$ is also prime.

N – $N = pq$, and has either m or $m-1$ bits.

h, x – elements of $\{1, \ldots, N-1\}$ (quadratic residues modulo N).

e' – a 161-bit prime number.

a – an element of $\{0, \ldots, (p-1)(q-1)/4 - 1\}$.

k' – element of $\mathbf{B}^{184}$.

s – element of $\mathbf{B}^{32}$.

5.2 Key Generation

Algorithm 5.2.1 generates an *ACE* signature key pair.

Algorithm 5.2.1 *Key generation for the ACE public-key signature scheme.*

Input: A size parameter $1024 \leq m \leq 16,384$.

Output: A public key/private key pair, as described in §5.1.

1. Generate random prime numbers p, q such that $(p-1)/2$ and $(q-1)/2$ are prime, and
$$2^{m_1-1} < p < 2^{m_1} \quad , \quad 2^{m_2-1} < q < 2^{m_2}, \quad \text{and} \quad p \neq q,$$
where
$$m_1 = \lfloor m/2 \rfloor \quad \text{and} \quad m_2 = \lceil m/2 \rceil.$$
2. Set $N \leftarrow p \cdot q$.

3. Generate a random prime number e', where $2^{160} < e' < 2^{161}$.
4. Generate $h' \in \{1, \ldots, N-1\}$ at random, subject to $\gcd(h', N) = 1$ and $\gcd(h' \pm 1, N) = 1$, and compute $h \leftarrow (h')^{-2} \text{ rem } N$.
5. Generate $a \in \{0, \ldots, (p-1)(q-1)/4 - 1\}$ at random, and compute $x \leftarrow h^a \text{ rem } N$.
6. Generate random byte strings $k' \in \mathbf{B}^{184}$, and $s \in \mathbf{B}^{32}$.
7. Return the public key/private key pair

$$((N, h, x, e', k', s), (p, q, a)).$$

5.3 Signature Representation

Consider an *ACE* signature public key (N, h, x, e', k', s), as described in §5.1. A signature of the *ACE* signature scheme has the form $(d, w, y, y', \tilde{k})$, where the components are as follows:

d – an element of $\mathbf{B}^{64}$.

w – an integer such that $2^{160} < w < 2^{161}$.

y, y' – elements of $\{1, \ldots, N-1\}$.

$\tilde{k}$ – an element of $\mathbf{B}^*$; note that $L(\tilde{k}) = 64 + 20L_{\mathbf{b}}(\lceil (L(M) + 8)/64 \rceil)$, where M is the message being signed.

We introduce the function *SEncode* that is used to map a signature to its byte-string representation, and the inverse function *SDecode*. For integer $l > 0$, byte string $d \in \mathbf{B}^{64}$, integers $0 \leq w < 256^{21}$, and $0 \leq y, y' < 256^l$, and byte string $\tilde{k} \in \mathbf{B}^*$,

$$\mathit{SEncode}(l, d, w, y, y', \tilde{k}) \stackrel{\text{def}}{=} d \,\|\, \mathit{pad}_{21}(I_{\mathbf{Z}}^{\mathbf{B}^*}(w)) \,\|\, \mathit{pad}_l(I_{\mathbf{Z}}^{\mathbf{B}^*}(y)) \,\|\, \mathit{pad}_l(I_{\mathbf{Z}}^{\mathbf{B}^*}(y')) \,\|\, \tilde{k} \quad \in \mathbf{B}^*.$$

For integer $l > 0$ and byte string $\sigma \in \mathbf{B}^*$ with $L(\sigma) \geq 53 + 2l$,

$$\begin{aligned} \mathit{SDecode}(l, \sigma) &\stackrel{\text{def}}{=} ([\sigma]_0^{64}, I_{\mathbf{B}^*}^{\mathbf{Z}}([\sigma]_{64}^{85}), I_{\mathbf{B}^*}^{\mathbf{Z}}([\sigma]_{85}^{85+l}), I_{\mathbf{B}^*}^{\mathbf{Z}}([\sigma]_{85+l}^{85+2l}), [\sigma]_{85+2l}^{L(\sigma)}) \\ &\in \mathbf{B}^{64} \times \mathbf{Z} \times \mathbf{Z} \times \mathbf{Z} \times \mathbf{B}^*. \end{aligned}$$

5.4 Signature Generation Operation

Algorithm 5.4.1 uses an *ACE* signature key pair to digitally sign messages.

Algorithm 5.4.1 *ACE signature generation.*

Input: A public key (N, h, x, e', k', s) and corresponding private key (p, q, a) as described in §5.1, and a byte string $M \in \mathbf{B}^*$, $0 \leq L(M) < 2^{64}$.

Output: A byte-string encoded signature $\sigma \in \mathbf{B}^*$ of M, as described in §5.3.

1. Perform the following steps to hash the input data:

1.1 Generate a hash key $\tilde{k} \in \mathbf{B}^{20m+64}$ at random, such that $m = L_{\mathbf{b}}(\lceil (L(M) + 8)/64 \rceil)$.

1.2 Compute $m_h \leftarrow I^{\mathbf{Z}}_{\mathbf{W}^*}(UOWHash''(\tilde{k}, M))$ (using Algorithm 5.6.1).

2. Select $\tilde{y} \in \{1, \ldots, N-1\}$ at random, and compute $y' \leftarrow \tilde{y}^2 \text{ rem } N$.

3. Compute $x' \leftarrow (y')^{e'} h^{m_h} \text{ rem } N$.

4. Generate a random prime e, $2^{160} < e < 2^{161}$, and its certificate of correctness (w, d) using Algorithm 5.5.1: $(e, w, d) \leftarrow GenCertPrime(s)$. Repeat this step until $e \neq e'$.

5. Set $r \leftarrow UOWHash'''(k', L_{\mathbf{B}}(N), x', \tilde{k}) \ \in \mathbf{Z}$ (using Algorithm 5.6.2); note that $0 \leq r < 2^{160}$.

6. Compute $y \leftarrow h^b \text{ rem } N$, where

$$b \leftarrow e^{-1}(a - r) \text{ rem } (p'q'),$$

and where $p' = (p-1)/2$ and $q' = (q-1)/2$.

7. Encode the signature as described in §5.3:

$$\sigma \leftarrow SEncode(L_{\mathbf{B}}(N), d, w, y, y', \tilde{k}).$$

8. Return σ.

5.5 Certified prime generation

The prime generation operation that is applied in Algorithm 5.4.1 produces a certified prime e of the form $2PR + 1$, $2^{160} < e < 2^{161}$, with a prime P, $2^{52} < P < 2^{53}$, and an integer R. Additionally, a certificate of correctness is generated which not only guarantees that e is prime, but also that e was generated in a highly constrained fashion.

Algorithm 5.5.1 *Certified prime generation GenCertPrime.*

Input: A byte string $s \in \mathbf{B}^{32}$.

Output: The tuple $(e, w, d) \in \mathbf{Z} \times \mathbf{Z} \times \mathbf{B}^{64}$—$2^{160} < e < 2^{161}$ and e is prime; $0 < w < e$ and w acts as a "witness" to the primality of e; and d acts as a "proof" that e was generated in a specific way.

1. Initialize $s_1 \leftarrow I^{\mathbf{W}^*}_{\mathbf{B}^*}([s]_0^{16}) \ \in \mathbf{W}^4$, $s_2 \leftarrow I^{\mathbf{W}^*}_{\mathbf{B}^*}([s]_{16}^{32}) \ \in \mathbf{W}^4$.

2. Generate a prime P, $2^{52} < P < 2^{53}$:

2.1 Generate $d_P \in \mathbf{B}^{32}$ at random, and compute

$$v_P \leftarrow I^{\mathbf{Z}}_{\mathbf{W}^*}(MARS(I^{\mathbf{W}^*}_{\mathbf{B}^*}(d_P), s_1) \oplus MARS(I^{\mathbf{W}^*}_{\mathbf{B}^*}(d_P), s_1 + 1)).$$

2.2 Compute a candidate integer P, $2^{52} < P < 2^{53}$: $P \leftarrow (v_P \text{ rem } 2^{52}) + 2^{52}$.

2.3 Test if P is prime by first performing some trial division, and then performing Miller-Rabin tests to the bases $2, 3, 5, 7, 11, 13, 23$; if P is not prime, then go to step 2.1.

3. Generate random $R \in \mathbf{Z}$ such that $2^{160} < 2PR + 1 < 2^{161}$:

   3.1 Select $d_R \in \mathbf{B}^{32}$ at random, and compute

   $$v_R \leftarrow I^{\mathbf{Z}}_{\mathbf{W}^*}(MARS(I^{\mathbf{W}^*}_{\mathbf{B}^*}(d_R), s_2) \oplus MARS(I^{\mathbf{W}^*}_{\mathbf{B}^*}(d_R), s_2 + 1)).$$

   3.2 Set $lb \leftarrow \lfloor (2^{160} - 1)/2P \rfloor$, $ub \leftarrow \lfloor (2^{161} - 1)/2P \rfloor$, and $bnd \leftarrow ub - lb$.

   3.3 If $v_R - (v_R \text{ rem } bnd) + bnd > 2^{128}$ then go to step 3.1.

   3.4 Set $R \leftarrow lb + (v_R \text{ rem } bnd) + 1$.

4. Set $e \leftarrow 2PR + 1$.

5. Test if e is divisible by small primes; if so, go to step 3.

6. Set $w \leftarrow 2$.

7. $status \leftarrow EvalPWitness(P, R, w)$ (see Algorithm 5.5.2).

8. If $status = \textsf{Reject}$, then generate random $w \in \{1, \ldots, e - 1\}$ and go to step 7; otherwise, if $status = \textsf{Composite}$, then go to step 3.

9. Set $d \leftarrow d_P \,\|\, d_R \in \mathbf{B}^{64}$.

10. Return (e, w, d).

Algorithm 5.5.2 *Prime witness evaluation EvalPWitness.*

Input: A tuple (P, R, w), where P is a prime such that $2^{52} < P < 2^{53}$, R is a positive integer such that that $2^{160} < 2PR + 1 < 2^{161}$, and w is an integer with $0 < w < 2PR + 1$.

Output: $status \in \{\textsf{Prime}, \textsf{Composite}, \textsf{Reject}\}$—if $status = \textsf{Prime}$, then $2PR+1$ is prime; if $status = \textsf{Composite}$, then $2PR + 1$ is composite; if $status = \textsf{Reject}$, then $2PR + 1$ may be either prime or composite.

1. Evaluate the candidate witness w:

   1.1 Set $e \leftarrow 2PR + 1$.

   1.2 If w is a Miller-Rabin witness to the compositeness of e, then return Composite.

   1.3 If $\gcd(w^{2R} - 1, e) \neq 1$, then return Reject.

2. Check if P and R satisfy the following conditions:

   2.1 If $R \not\equiv m \pmod{2Pm + 1}$ for all integers m such that $1 \leq m < e/(4P^3)$, then return Composite; note that $e/(4P^3) < 8$.

   2.2 Let x, y be integers such that $R = 2Px + y$ and $0 \leq y < 2P$; if $y^2 - 4x = z^2$ for some $z \in \mathbf{Z}$, then return Composite.

3. Return Prime.

5.6 *UOWHash* variants with length encoding and padding

First comes function $UOWHash''$, which pads and encodes the length of the input before calling *UOWHash*.

Algorithm 5.6.1 *Universal one-way hash function $UOWHash''$.*

Input: A tuple $(k, M) \in \mathbf{B}^* \times \mathbf{B}^*$, where $L(k) = 20m + 64$ for some integer $m \geq L_{\mathbf{b}}(\lceil (L(M) + 8)/64 \rceil)$, and $0 \leq L(M) < 2^{64}$.

Output: The hash value $h \in \mathbf{W}^5$ of a padded, length encoded version of M under key k.

1. Pad M to obtain a byte string M' whose length is a multiple of 64, and where the last 8 bytes of M' encode $L(M)$:
$$M' \leftarrow pad_{l-8}(M) \,\|\, pad_8(I_{\mathbf{Z}}^{\mathbf{B}^*}(L(M))) \;\; \in \mathbf{B}^l,$$
where $l = 64\lceil (L(M) + 8)/64 \rceil$.
2. Compute
$$h \leftarrow UOWHash(I_{\mathbf{B}^*}^{\mathbf{W}^*}(k), I_{\mathbf{B}^*}^{\mathbf{W}^*}(M')) \in \mathbf{W}^5.$$
3. Return h.

Next comes function $UOWHash'''$, which is a special-purpose hash function used in the signature scheme.

Algorithm 5.6.2 *Universal one-way hash function $UOWHash'''$.*

Input: A tuple $(k', l, x', \tilde{k}) \in \mathbf{B}^* \times \mathbf{Z} \times \mathbf{Z} \times \mathbf{B}^*$, where $l \geq 0$, $0 \leq x' < 256^l$, and $L(k') = 20m + 64$ for some $m \geq 0$ such that $m \geq L_{\mathbf{b}}(\lceil (l' + 8)/64/) \rceil$ and $l' < 2^{64}$, where $l' = 4\lceil l/4 \rceil + L(\tilde{k})$.

Output: The hash value $r \in \mathbf{Z}$ (with $0 \leq r < 2^{160}$) of a padded, length encoded version of $(x', \tilde{k})$ under key k'.

1. Set $k_h \leftarrow pad_{l_1}(I_{\mathbf{Z}}^{\mathbf{B}^*}(x')) \,\|\, \tilde{k}$, where $l_1 = 4\lceil l/4 \rceil$.
2. Set $r \leftarrow I_{\mathbf{W}^*}^{\mathbf{Z}}(UOWHash''(k', k_h))$ (using Algorithm 5.6.1).
3. Return r.

5.7 Signature Verification Operation

Algorithm 5.7.1 uses an *ACE* public key to verify a signature with respect to a given message.

Algorithm 5.7.1 *ACE signature verification.*

Input: A public key (N, h, x, e', k', s) as described in §5.1, a signature $\sigma \in \mathbf{B}^*$, and a message $M \in \mathbf{B}^*$.

Output: $status \in \{\mathsf{Accept}, \mathsf{Reject}\}$—if σ is a valid signature on M under the given public key, then $status = \mathsf{Accept}$; otherwise, $status = \mathsf{Reject}$.

1. Decode the signature as described in §5.3:
   1.1 If $L(M) \geq 2^{64}$ then stop processing and signal Reject.
   1.2 If $L(\sigma) < 85 + 2L_{\mathbf{B}}(N)$ then stop processing and signal Reject.
   1.3 Compute
   $$(d, w, y, y', \tilde{k}) \leftarrow SDecode(L_{\mathbf{B}}(N), \sigma) \in \mathbf{B}^{64} \times \mathbf{Z} \times \mathbf{Z} \times \mathbf{Z} \times \mathbf{B}^*;$$
   note that $0 \leq w < 256^{21}$ and $0 \leq y, y' < 256^l$, where $l = L_{\mathbf{B}}(N)$.
2. Set $e \leftarrow VerCertPrime(s, d, w)$ (using Algorithm 5.7.2).
3. If $e =$ Reject, return Reject.
4. If $e = e'$, then return Reject.
5. If $y = 0$ or $y \geq N$ or
   $y' = 0$ or $y' \geq N$ then return Reject.
6. Perform the following steps to hash the input data:
   6.1 If $L(\tilde{k}) \neq 20m + 64$, where $m = L_{\mathbf{b}}(\lceil (L(M)+8)/64 \rceil)$, then return Reject.
   6.2 Compute $m_h \leftarrow I^{\mathbf{Z}}_{\mathbf{W}^*}(UOWHash''(\tilde{k}, M))$ (using Algorithm 5.6.1).
7. Compute $x' \leftarrow (y')^{e'} h^{m_h} \text{ rem } N$.
8. Set $r \leftarrow UOWHash'''(k', L_{\mathbf{B}}(N), x', \tilde{k}) \in \mathbf{Z}$ (using Algorithm 5.6.2); note that $0 \leq r < 2^{160}$.
9. If $x \not\equiv y^e h^r \pmod{N}$ then return Reject.
10. Return Accept.

The certificate verification operation that is applied in Algorithm 5.7.1 checks whether a presented integer witnesses the primality of a candidate prime of a certain form, given by its descriptor.

Algorithm 5.7.2 *Prime certificate verification VerCertPrime.*

Input: The tuple (s, d, w) containing byte strings $s \in \mathbf{B}^{32}, d \in \mathbf{B}^{64}$, and an integer $w \geq 0$.

Output: A prime e derived from s and d, with $2^{160} < e < 2^{161}$, or the symbol Reject.

1. Initialize $s_1 \leftarrow I^{\mathbf{W}^*}_{\mathbf{B}^*}([s]_0^{16}) \in \mathbf{W}^4$, $s_2 \leftarrow I^{\mathbf{W}^*}_{\mathbf{B}^*}([s]_{16}^{32}) \in \mathbf{W}^4$,
   $d_P \leftarrow I^{\mathbf{W}^*}_{\mathbf{B}^*}([d]_0^{32}) \in \mathbf{W}^8$, $d_R \leftarrow I^{\mathbf{W}^*}_{\mathbf{B}^*}([d]_{32}^{64}) \in \mathbf{W}^8$.
2. Compute and validate prime P:
   2.1 Compute $v_P \leftarrow I^{\mathbf{Z}}_{\mathbf{W}^*}(MARS(d_P, s_1) \oplus MARS(d_P, s_1 + 1))$.

2.2 Set $P \leftarrow (v_P \text{ rem } 2^{52}) + 2^{52}$.

2.3 Test if P is prime by performing Miller-Rabin tests to the bases $2, 3, 5, 7, 11, 13, 23$; if P is not prime, then return Reject.

3. Compute and validate the coefficient R:

3.1 Compute $v_R \leftarrow I^{\mathbf{Z}}_{\mathbf{W}^*}(MARS(d_R, s_2) \oplus MARS(d_R, s_2 + 1))$.

3.2 Set $lb \leftarrow \lfloor (2^{160} - 1)/2P \rfloor$, $ub \leftarrow \lfloor (2^{161} - 1)/2P \rfloor$, and $bnd \leftarrow ub - lb$.

3.3 If $v_R - (v_R \text{ rem } bnd) + bnd > 2^{128}$, then return Reject.

3.4 Set $R \leftarrow lb + (v_R \text{ rem } bnd) + 1$.

4. Set $e \leftarrow 2PR + 1$.

5. If $w = 0$ or $w \geq e$ return Reject.

6. If $EvalPWitness(P, R, w) \neq$ Prime (see Algorithm 5.5.2), then return Reject.

7. Return e.

5.8 Security analysis

We briefly summarize the security properties of the above signature scheme. The bulk of the analysis already appears in [CS99]. We simply fill in the details here.

Consider an adversary that runs in time at most t, makes at most κ signature requests, with the total byte length of these messages being at most l. The adversary's advantage, $\mathsf{AdvEnc}(t, \kappa, l)$ (as defined in §2.3) can be computed in terms of

- the advantage the adversary has in breaking the RSA and strong RSA assumptions (see AdvRSA and AdvFlexRSA, defined in §2.6), the advantage the adversary has in finding second preimages in SHA-1 (see AdvSHA, defined in §2.7), and
- the advantage the adversary has in distinguishing MARS output from random (see AdvMARS, defined in 2.8).

Also, we let $l' = L_{\mathbf{B}}(N)$, let T'_e be the time required for a 161-bit exponentiation, modulo a 161-bit number, and let T_e be the time required for a 161-bit exponentiation modulo N.

Theorem 5.8.1 *Assuming the Generalized Riemann Hypothesis, we have:*

$$\begin{aligned} \mathsf{AdvSig}(t, \kappa, l) \quad \leq \quad & \mathsf{AdvRSA}(O(t + T_e \kappa \log \kappa)) \cdot (\kappa + 1) + \\ & \mathsf{AdvFlexRSA}(O(t + T_e \kappa \log \kappa) \cdot 1.01 + \\ & \mathsf{AdvSHA}(O(t)) \cdot \left\{ \kappa \left(90 + \frac{l'}{64} \right) + \frac{l}{64} \right\} + \\ & \mathsf{AdvMARS}(O(T'_e \kappa), 1) \cdot (2^{16} + 150\kappa) + \\ & \kappa^2 / 2^{145} + \\ & 2^{-80}. \end{aligned}$$

Call the original attack game G_0. Let S_0 be the event that the adversary forges a signature in this game. We have

$$\mathsf{AdvSig}(t, \kappa, l) = \Pr[S_0]. \tag{24}$$

We shall make two transformation of this game, obtaining games G_1, G_2. In order to relate probabilities of events in different games, conceptually, these games are all run on the same underlying probability distribution. In each game G_i, for $i = 1, 2$, we let S_i denote the event that the adversary forges a signature in game G_i.

Game G_1. Let U_0 be the event that the adversary in game G_0, the adversary presents a forged signature σ' such that either

- U_0^1 : the hash computed in step 8 of Algorithm 5.7.1, when applied to σ', yields a non-trivial collision with one of the hashes computed in step 8 of Algorithm 5.7.1, when applied to some signature σ created by the signing algorithm, or
- U_0^2 : the key $\tilde{k}$ in σ' matches that of one of the signatures σ created by the signing algorithm, and the hash computed in step 6 of Algorithm 5.7.1, when applied to σ', yields a collision with the hash computed in step 1 of Algorithm 5.7.1, when applied to σ.

Game G_1 is just like game G_0, except that should event U_0 occur, we stop the game without allowing the forgery to be presented.

One can show that

$$\Pr[U_0^1] \leq \mathsf{AdvSHA}(O(t)) \cdot \kappa(88 + l'/64). \tag{25}$$

This is obtained by using the analysis in [Sho00a], plus a "plug and pray" argument. We guess on which of κ signatures this collision will occur, and the position of the "target" block, i.e., on which 512-bit hash input block the collision will occur. Moreover, because the hash inputs under consideration can vary in length, we have to guess whether the target block is the last block of the hash input, and if it is the last block, we have to guess exactly how many 160-bit masks (comprising $\tilde{k}$) there actually are (there are at most three choices, given that the target block is the last input block). Making these guesses, and given an instance of the second preimage problem, we generate an appropriate *prefix* of the hash input, from which we can generate the corresponding key k' using the key reconstruction algorithm in [Sho00a]. An important feature of of the key reconstruction algorithm in [Sho00a] that we exploit here is that it relies only on the prefix of the hash input up to, and including, the target input block. The adversary's view is independent of these guesses, and if these guesses are correct, then we solve the given second preimage problem.

Note that the above argument is a bit complicated, but it gives a numerically much better result than the simpler, and more generic "plug and pray" argument where we guess the signature, the length of the input to the hash function, and the position of the target block.

One can also show that

$$\Pr[U_0^2] \leq \mathsf{AdvSHA}(O(t)) \cdot (l + 2\kappa). \tag{26}$$

This is also obtained by using the analysis in [Sho00a], plus a "plug and pray" argument. The quantity $l + 2\kappa$ is a bound on the total number of relevant hash input blocks, and we have to guess which of these is the "target" block.

It is clear that

$$\Pr[S_1|U_0] = \Pr[S_0|U_0], \tag{27}$$

and hence we can apply Lemma 4.10.1 with $(E, E', F, F') = (S_0, S_1, U_0, U_0)$, obtaining

$$\Pr[S_0] \le \Pr[S_1] + \mathsf{AdvSHA}(O(t)) \cdot \left\{\kappa\left(90 + \frac{l'}{64}\right) + \frac{l}{64}\right\}. \tag{28}$$

Game G_2. This game is just like game G_1, except for the way in which the primes e generated by the signing algorithm are generated. Define $b_P = 2^{14} + 38\kappa$, $b_R = 2^{15} + 112\kappa$, and $b_w = 2^{17} + 448\kappa$. In game G_2, we generate κ primes in advance, to be used later by the signing algorithm. We use Algorithm 5.5.1 to generate primes as in game G_1. However, in this game, we stop if the event V that one of the following occurs:

- step 2.1 in Algorithm 5.5.1 is executed more than b_P times,
- step 3.1 in Algorithm 5.5.1 is executed more than b_R times,
- step 7 is Algorithm 5.5.1 is executed more than b_w times, or
- two of the generated primes are equal.

Let V' be the corresponding event, but where the strings v_P and v_R generated in Algorithm 5.5.1 are truly random. Then we have

$$\begin{aligned}\Pr[V] &\le \Pr[V'] + \mathsf{AdvMARS}(O(T'_e\kappa), 1) \cdot (b_P + b_R) \\ &\le \kappa^2/2^{145} + 2^{-80} + \mathsf{AdvMARS}(O(T'_e\kappa), 1) \cdot (b_P + b_R). \end{aligned} \tag{29}$$

The term 2^{-80} comes from a calculation using Chernoff's bound together with prime density estimates used in [CS99]. The term $\kappa^2/2^{145}$ also comes from the prime density estimates used in [CS99]. Both of these density estimates rely on the Generalized Riemann Hypothesis.

Again applying Lemma 4.10.1, we see that

$$\Pr[S_1] \le \Pr[S_2] + \kappa^2/2^{145} + 2^{-80} + \mathsf{AdvMARS}(O(T'_e\kappa), 1) \cdot (b_P + b_R). \tag{30}$$

Note that the running time of game G_2 is $O(t + T'_e\kappa)$.

Now, appealing to the proof of security in [CS99], and using a careful implementation of the simulators in that paper, one can show that

$$\Pr[S_2] \le \mathsf{AdvRSA}(O(t + T_e\kappa\log\kappa)) \cdot (\kappa + 1) + \mathsf{AdvFlexRSA}(O(t + T_e\kappa\log\kappa) \cdot 1.01. \tag{31}$$

The term $T_e\kappa\log\kappa$ in the above running times deserves some comment. In the simulators described in [CS99], at a couple of points, we have to perform a computation of the following type. Let $e_1, \ldots, e_\kappa$ be the primes generated by the signing algorithm, and let $E = \prod_{i=1}^{\kappa} e_i$. Given $w \in \{0, \ldots, N-1\}$, we have to compute $w^{E/e_i} \operatorname{rem} N$ for $1 \le i \le \kappa$. Naively, one could do this in time $O(T_e\kappa^2)$. However, using a simple divide-and-conquer algorithm (see, e.g., §6 of [Sho94]), one can do this in time $O(T_e\kappa\log\kappa)$.

The theorem now follows from (24), (28), (30), and (31).

5.9 Further discussion and implementation notes

Optimizations

In Algorithm 5.4.1, the exponentiations performed in steps 3 and 6 are well-suited for optimization. First, since the signer knows the factorization of N, one may use the Chinese Remainder Theorem to speed up the computation. Also, in step 3, we need to compute the product of two powers, which can be performed using standard algorithmic techniques significantly faster than two independent exponentiations. And in step 6, we need to raise h to a power. Since h depends on the public key, we can condition on h, so that raising h to a power can be done significantly faster than an ordinary exponentiation. In Algorithm 5.7.1, in steps 7 and 9, we also need to compute products of powers, which are subject to standard optimizations as above. We refer the reader to §14.6 of [MvOV97] for details of all of these optimizations.

The multi-user setting

At least in an asymptotic sense, the definition of security we have used implies security in a multi-user environment. Using a standard "hybrid" argument, one sees that security essentially degrades by a factor portional to the number of users.

We believe that our choices of parameters allow sufficient "head room" so that one still obtains a meaninful level of security even considering fairly large systems of users. However, an even higher level of security could be obtained with some modification to the basic algorithms. This would lead to somewhat more complicated algorithms, and would require all users to share the same UOWH key, which introduces a "trust" problem. For these reasons, we have not chosen to pursue this at the moment.

Implementation of the key generation algorithm

In the key generation step, we have to generate "strong primes" of the form $p = 2p' + 1$, where p' is also prime. The number p' is also known as a Sophie Germain prime. This can be a fairly time-consuming computation, and some care must be taken to use an efficient algorithm for this task.

The most naive way to do this is to generate a prime p', and then test if $2p' + 1$ is also prime. However, we do not recommend this approach. Rather, we recommend the approach descrined in the full-length version of [CS99], which can easily yield a factor of 10 speed-up over the naive method.

API considerations

We have designed the signing and verification algorithms so that they can work with *streams* of data. Both the signing and verification algorithm can process the message as a stream. However, the verification algorithm needs the to have the signature *before* processing the message stream. This is a bit non-standard, and in some situations may be a bit awkward. For most signature schemes used in practice, the verification algorithm can process the message as a stream, requiring the signature only *after* the message stream has been processed. The reason our verification algorithm needs the signature first is that it needs the key $\tilde{k}$ to the universal one-way hash function used to hash the message. This seems unavoidable if we want to use universal one-way hash functions instead of collision resistant hash functions, which—as we have already argued—is quite desirable from a security point of view. One partial solution to the problem would be to have the signer generate a key $\tilde{k}$ of sufficient length before processing its message input stream, placing $\tilde{k}$ in its output stream before placing any of the message bytes in its output stream. This would allow the signer's output stream to be bound directly to the verifier's input stream, without requiring any significant buffering on the part of either the signer or verifier. However, the resulting interface would still be somewhat non-standard.

Random oracles

Although we use the strong RSA assumption, the form of the strong RSA assumption we actually use severely constrains the adversary's behavior: it is not free to choose an exponent e as it pleases, but rather, it must choose $e = 2PR + 1$, where both P and R are computed as the output of a one-way cryptographic transformation. As already mentioned in 2.4, in the random oracle model, our signature scheme can be proved secure under the RSA assumption, instead of the strong RSA assumption. Actually, to be a bit more precise, we need to use the *ideal cipher model* (see [KR96]), which is a closely related, but slightly different model of analysis. This is discussed in [CS99].

6 ASN.1 Key Syntax

For applications that use ASN.1 descriptions, like for example X.509 or PKCS#8 key formats, it is necessary to define the algorithm identifier for the schemes defined in this document, along with their key types. However, the corresponding object identifiers are not defined yet, let alone registered. There are no parameters used, hence, the associated parameters field of the algorithm identifier is of type NULL.

```
Version ::= INTEGER
```

The version number is for compatibility with future revisions of this document. It shall be 1 for this version of the document.

6.1 Encryption Key Pair

This section defines the ASN.1 types ACEEncPubKey and ACEEncPrivKey. The corresponding fields as described in §4.1 are given in comments.
An *ACE* encryption public key should be represented as follows:

```
ACEEncPubKey ::= SEQUENCE {
  version Version,
  prime1 INTEGER,          -- P
  prime2 INTEGER,          -- q
  num1 INTEGER,            -- g1
  num2 INTEGER,            -- g2
  num3 INTEGER,            -- c
  num4 INTEGER,            -- d
  seed1 INTEGER,           -- h1
```

```
  seed2 INTEGER,          -- h2
  hkey1 OCTET STRING,     -- k1
  hkey2 OCTET STRING      -- k2
}
```

An *ACE* encryption private key should be represented as the following ASN.1 type:

```
ACEEncPrivKey ::= SEQUENCE {
  version Version,
  prime1 INTEGER,         -- P
  prime2 INTEGER,         -- q
  num1 INTEGER,           -- w
  num2 INTEGER,           -- x
  num3 INTEGER,           -- y
  num4 INTEGER,           -- z1
  num5 INTEGER,           -- z2
  hkey1 OCTET STRING,     -- k1
  hkey2 OCTET STRING      -- k2
}
```

Note that unlike in §4.1, this structure defines a "self contained" key—the decryption algorithm needs only the data in this structure, and does need need any of the data in the structure describing the public key.

6.2 Signature Key Pair

This section defines the ASN.1 types ACESigPubKey and ACESigPrivKey. The corresponding fields as described in §5.1 are given in comments.
An *ACE* signature public key should be represented as follows:

```
ACESigPubKey ::= SEQUENCE {
  version Version,
  modulus INTEGER,         -- N
  num1 INTEGER,            -- h
  num2 INTEGER,            -- x
  primeExp INTEGER,        -- e'
  hkey OCTET STRING,       -- k'
  primeParam OCTET STRING  -- s
}
```

An *ACE* signature private key should be represented as the following ASN.1 type:

```
ACESigPrivKey ::= SEQUENCE {
  version Version,
  modulus INTEGER,         -- N
  prime1 INTEGER,          -- p
  prime2 INTEGER,          -- q
  auxExp INTEGER,          -- a
  num1 INTEGER,            -- h
  primeExp INTEGER,        -- e'
  hkey OCTET STRING,       -- k'
  primeParam OCTET STRING  -- s
}
```

	Power PC		Pentium	
	operand size (bytes)		operand size (bytes)	
	512	1024	512	1024
multiplication	$3.5 \times 10^{-5}s$	$1.0 \times 10^{-4}s$	$4.5 \times 10^{-5}s$	$1.4 \times 10^{-4}s$
squaring	$3.3 \times 10^{-5}s$	$1.0 \times 10^{-4}s$	$4.4 \times 10^{-5}s$	$1.4 \times 10^{-4}s$
exponentiation	$1.9 \times 10^{-2}s$	$1.2 \times 10^{-1}s$	$2.6 \times 10^{-2}s$	$1.7 \times 10^{-1}s$

Table 1: Times for basic operations

	Power PC		Pentium	
	Fixed costs (ms)	Mbits/sec	Fixed costs (ms)	Mbits/sec
encrypt	160	18	230	16
decrypt	68	18	97	14
sign	48	64	62	52
sign set-up	29		41	
verify	52	65	73	53

Table 2: Encryption and signature scheme performance

Note that unlike in §5.1, this structure defines a "self contained" key—the signing algorithm needs only the data in this structure, and does need need any of the data in the structure describing the public key.

7 Performance

We report here on the performance of an implementation of our encryption and signature scheme.

We implemented both schemes in ANSI C, using the GNU GMP library to implement the multi-precision arithmetic, although we implemented our own "sliding window" exponentiation routine, as this was not provided in GMP.

We performed timing experiments on two platforms. The first platform is a PowerPC 604 model 43P processor running AIX. The second platform is a 266MHz Pentium running Windows NT.

As a baseline, we first report the times for 512-bit and 1024-bit multiplication, squaring, and exponentiation in Table 1.

Table 2 reports the performance of the encryption and signature schemes. For both schemes, a 1024-bit modulus was used. In reporting the time to sign a message, we break the fixed-cost time into two components. One component is the "sign set-up" time, which is the time to perform a pre-computation that depends only on the secret key; if many signatures are to be generated using a given key, the "sign set-up" operation need be executed only once. The other component is the "sign" time, which is the time to generate a signature using the data computed in the "sign set-up" operation. We also mention that roughly one third of the "sign" time is spent generating the required 161-bit prime. For larger moduli, this time take a smaller proportion of the whole.

Finally, we mention the time required to generate public keys (again, with 1024-bit moduli). The key generation algorithm for our signature scheme is a bit unusual, since it requires the generation of primes of the form $2p' + 1$, where p' is also prime. This can be quite costly, and as already mentioned, some care must be taken in the implementation of this step.

In our implementation, on the PowerPC platform, the average time for the signature key generation algorithm is $35s$, and the average time for the encryption key generation algorithm is $11s$. On the Pentium platform, the corresponding times were $36s$ and $14s$, respectively.

References

[ABR98] M. Abdalla, M. Bellare, and P. Rogaway. DHAES: an encryption scheme based on the Diffie-Hellma problem. Submission to IEEE P1363, 1998.

[BBM00] M. Bellare, A. Boldyreva, and S. Micali. Public-key encryption in a multi-user setting: security proofs and improvements. In *Advances in Cryptology–Eurocrypt 2000*, 2000.

[BCD+98] C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S. Matyas Jr., L. O'Connor, M. Peyravian, D. Safford, and N. Zunic. MARS–a candidate cipher for AES, June 1998.

[BI99] M. Bellare and R. Impagliazzo. A tool for obtaining tighter security analyses of pseudorandom function based constructions, with applications to PRP $\rightarrow$ PRF conversion. Manuscript, 1999.

[Ble98] D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In *Advances in Cryptology–Crypto '98*, pages 1–12, 1998.

[Bon98] D. Boneh. The Decision Diffie-Hellman Problem. In *Ants-III*, pages 48–63, 1998. Springer LNCS 1423.

[BP97] N. Barić and B. Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In *Advances in Cryptology–Eurocrypt '97*, pages 480–494, 1997.

[BR93] M. Bellare and P. Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *First ACM Conference on Computer and Communications Security*, pages 62–73, 1993.

[BR94] M. Bellare and P. Rogaway. Optimal asymmetric encryption. In *Advances in Cryptology—Crypto '94*, pages 92–111, 1994.

[Bra93] S. Brands. An efficient off-line electronic cash system based on the representation problem, 1993. CWI Technical Report, CS-R9323.

[BS96] E. Bach and J. Shallit. *Algorithmic Number Theory*, volume 1. MIT Press, 1996.

[CGH98] R. Canetti, O. Goldreich, and S. Halevi. The random oracle model, revisted. In *30th Annual ACM Symposium on Theory of Computing*, 1998.

[CHJ99] D. Coppersmith, S. Halevi, and C. Jutla. ISO 9796-1 and the new forgery strategy. Unpublished manuscript, 1999.

[CNS99] J. Coron, D. Naccache, and J. Stern. On the security of RSA padding. In *Advances in Cryptology–Crypto '99*, pages 1–18, 1999.

[CS98] R. Cramer and V. Shoup. A practical public key cryptosystem provably secure against adaptive chosen ciphertext attack. In *Advances in Cryptology–Crypto '98*, pages 13–25, 1998.

[CS99] R. Cramer and V. Shoup. Signature schemes based on the strong RSA assumption. In *6th ACM Conf. on Computer and Communications Security*, 1999.

[Dam87] I. Damgård. Collision free hash functions and public key signature schemes. In *Advances in Cryptology–Eurocrypt '87*, 1987.

[dBB93] D. den Boer and A. Bosselaers. Collisions for the compression function of MD5. In *Advances in Cryptology–Eurocrypt '93*, pages 293–304, 1993.

[DDN91] D. Dolev, C. Dwork, and M. Naor. Non-malleable cryptography. In *23rd Annual ACM Symposium on Theory of Computing*, pages 542–552, 1991.

[DH76] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Trans. Info. Theory*, 22:644–654, 1976.

[DLP93] I. Damgård, P. Landrock, and C. Pomerance. Average case error estimates for the strong probable prime test. *Math. Comp.*, 61:177–194, 1993.

[Dob96] H. Dobbertin. The status of MD5 after a recent attack. *RSA Laboratories' CryptoBytes*, 2(2), 1996. The main result of this paper was announced at the Eurocrypt '96 rump session.

[DvOW92] W. Diffie, P. van Oorschot, and M. Wiener. Authentication and authenticated key exchange. *Designs, Code, and Cryptography*, 2:107–125, 1992.

[FO99] E. Fujisaki and T. Okamoto. Statistical zero knowledge protocols to prove modular polynomial relations. In *Advances in Cryptology–Crypto '97*, 1999.

[FS87] A. Fiat and A. Shamir. How to prove yourself: practical solutions to identification and signature problems. In *Advances in Cryptology–Crypto '86, Springer LNCS 263*, pages 186–194, 1987.

[GHR99] R. Gennaro, S. Halevi, and T. Rabin. Secure hash-and-sign signatures without the random oracle. In *Advances in Cryptology–Eurocrypt '99*, pages 123–139, 1999.

[GM84] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28:270–299, 1984.

[GMR88] S. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM J. Comput.*, 17:281–308, 1988.

[IZ89] R. Impagliazzo and D. Zuckermann. How to recycle random bits. In *30th Annual Symposium on Foundations of Computer Science*, pages 248–253, 1989.

[KJJ99] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in Cryptology–Crypto '99*, pages 388–397, 1999.

[Koc96] P. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology–Crypto '96*, pages 104–113, 1996.

[KR96] J. Kilian and P. Rogaway. How to protect DES against exhaustive key search. In *Advances in Cryptology–Crypto '96*, pages 252–267, 1996.

[Kra94] H. Krawczyk. LFSR-based hashing and authentication. In *Advances in Cryptology—Crypto '94*, pages 129–139, 1994.

[Lub96] M. Luby. *Pseudorandomness and Cryptographic Applications.* Princeton University Press, 1996.

[Luc00] S. Lucks. The sum of PRPs is a secure PRF. In *Advances in Cryptology–Eurocrypt 2000*, 2000.

[Mau99] U. Maurer. Information-theoretic cryptography. In *Advances in Cryptology–Crypto '99*, pages 47–64, 1999.

[MvOV97] A. Menesez, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997.

[NR97] M. Naor and O. Reingold. Number-theoretic constructions of efficient pseudo-random functions. In *38th Annual Symposium on Foundations of Computer Science*, 1997.

[NY89] M. Naor and M. Yung. Universal one-way hash functions and their cryptographic applications. In *21st Annual ACM Symposium on Theory of Computing*, 1989.

[NY90] M. Naor and M. Yung. Public-key cryptosystems provably secure against chosen ciphertext attacks. In *22nd Annual ACM Symposium on Theory of Computing*, pages 427–437, 1990.

[PS96] D. Pointcheval and J. Stern. Security proofs for signature schemes. In *Advances in Cryptology–Eurocrypt '96*, pages 387–398, 1996.

[RS91] C. Rackoff and D. Simon. Noninteractive zero-knowledge proof of knowledge and chosen ciphertext attack. In *Advances in Cryptology–Crypto '91*, pages 433–444, 1991.

[SHA95] Secure hash standard, National Institute of Standards and Technology (NIST), FIPS Publication 180-1, April 1995.

[Sho94] V. Shoup. Fast construction of irreducible polynomials over finite fields. *J. Symbolic Comp.*, 17(5):371–391, 1994.

[Sho98] V. Shoup. Why chosen ciphertext security matters. IBM Research Report RZ 3076, November 1998.

[Sho00a] V. Shoup. A composition theorem for universal one-way hash functions. In *Advances in Cryptology–Eurocrypt 2000*, 2000.

[Sho00b] V. Shoup. Using hash functions as a hedge against chosen ciphertext attack. In *Advances in Cryptology–Eurocrypt 2000*, 2000.

[Sim98] D. Simon. Finding collisions on a one-way street: can secure hash functions be based on general assumptions? In *Advances in Cryptology–Eurocrypt '98*, pages 334–345, 1998.

[Sta96] M. Stadler. Publicly verifiable secret sharing. In *Advances in Cryptology–Eurocrypt '96*, pages 190–199, 1996.

[ZS92] Y. Zheng and J. Seberry. Practical approaches to attaining security against adaptively chosen ciphertext attacks. In *Advances in Cryptology–Crypto '92*, pages 292–304, 1992.