Security Issues in the Diffie-Hellman Key Agreement Protocol

Jean-François Raymond and Anton Stiglic
Zero-Knowledge Systems, Inc.
{jfr, anton}@zeroknowledge.com

December 19, 2000

Abstract

Diffie-Hellman key agreement protocol [27] implementations have been plagued by serious security flaws. The attacks can be very subtle and, more often than not, haven't been taken into account by protocol designers. In this paper we attempt to provide a link between theoretical research and real-world implementations. In addition to exposing the most important attacks and issues we present fairly detailed pseudo-code for the authenticated Diffie-Hellman protocol and for the half-certified Diffie-Hellman (a.k.a. Elgamal key agreement). It is hoped that computer security practitioners will obtain enough information to build and design secure and efficient versions of this classic key agreement protocol.

Contents

1 Introduction

In their landmark 1976 paper "New Directions in Cryptography"[27], Diffie and Hellman present a secure key agreement protocol that can be carried out over public communication channels. Their protocol is still widely used to this day.

Even though the protocol seems quite simple, it *can* be vulnerable to certain attacks. As with many cryptographic protocols, the Diffie-Hellman key agreement protocol (DH protocol) has subtle problems that cryptographers have taken many years to discover. This vulnerability is compounded by the fact that programmers often don't have a proper understanding of the security issues. In fact, bad implementations of cryptographic protocols are, unfortunately, common [4].

In this work, we attempt to give a comprehensive listing of attacks on the DH protocol. This listing will in turn allow us to motivate protocol design decisions. Note that throughout this presentation emphasis is placed on practice. After reading this paper, one might not have an extremely detailed understanding of previous work and theoretical problems, but should have a very good idea about how to securely implement the DH protocol in different settings.

1.1 Related Work

As mentioned previously, flaws in cryptographic protocols are not uncommon. As this is a very important problem, it has received some attention; here are the most important approaches that have been proposed:

1. The use of verification logics such as BAN [18] to prove protocol properties.
2. Very high level programming languages in which security properties can be proved mechanically (i.e. by computers) [1].
3. Complete proofs of security [10, 9].
4. The use of robustness principles, i.e. rules of thumb, protocol design principles [5].

The biggest problem with the first approach is that encryption primitives are dissociated from the verification logics, which implies that they don't provide complete proofs of security [9]. As an example of this problem one just needs to look at the problem of encryption and signature ordering: most verification logics don't complain when messages are encrypted before being signed which possibly results in a security vulnerability [5].

The second approach seems very promising however the best known proof mechanization techniques aren't efficient enough and only a few cryptographic primitives have been included in the model.

The third suggestion is the most powerful. The main problem is that the proofs are somewhat involved and proving the correctness of complex protocols seems quite difficult. Note also that it's not entirely obvious that the claims that are proved are adequate.

The robustness principles are useful in that they can help in preventing common errors. However, it is impossible to exhaustively list all important robustness principles, and so using these principles doesn't give us peace of mind as there are no security guarantees. Furthermore, the protocol designer must be comfortable and competent in verifying security properties. For example Principle 3 of [5], which states

> *Be careful when signing or decrypting data that you never let yourself be used as an oracle*

might not be understood by individuals that don't have a background in cryptography.

The most important problem with all of the above approaches is that low level implementation issues aren't spelled out. Hence, unless one has a solid grasp of all of the details, it's easy to make low-level implementation errors. Also notice that none of these approaches deal with problems specific to the cryptographic primitives used.

Many standards have been developed for the DH protocol (see Appendix A), unfortunately none describe the issues and attacks in detail. More importantly, none motivate the design decisions. In [12], work has been done to characterize the security of the DH protocols introduced in various standards. We take a different, more general and perhaps more thorough approach by :

1. Describing and studying the most important theoretical and practical issues.
2. Presenting pseudo-code that implements secure protocols whose design is based on the issues we describe.

1.2 Overview

Section 2 presents a mathematical background of the basics needed to understand the DH protocol and the types of attacks it is vulnerable to. In section 3 we give attacks which are based on mathematical tricks. Authentication is discussed in section 4.3. In section 5 we discuss attacks on DH that exploit implementation details. In section 6, we expose some subtleties that appear when using the DH shared secret to obtain a key which can be used in other cryptographic operations. The information acquired in sections 3, 4.3, 5 and 6 is used to present implementation guidelines in section 7. In section 8, we present pseudo-code for an ephemeral authenticated DH protocol and for a half-certified DH protocol; we also describe all the primitives needed. The conclusion can be found in section 9.

2 The Diffie-Hellman Key Agreement Protocol

The DH key agreement protocol allows two users, referred to as Alice ($\mathcal{A}$) and Bob ($\mathcal{B}$), to obtain a shared secret key over a public communication channel. An attacker, *eavesdropping* at the messages sent by both Alice and Bob won't be able to determine what the shared secret key is. This is an extremely useful primitive because the shared secret can be used to generate a secret session key that can be used with symmetric crypto-systems[1] (e.g. DES) or message authentication codes (MAC). We now give some basic notions from mathematics that are needed to understand the protocol.

2.1 Mathematical Background

The computations required in the DH protocol are carried out in a group.

2.1.1 Groups

A *group* $(G, \star)$ consists of a set G and a binary operation $\star$ that takes elements of G as inputs. $\star$ has the following properties:

1. (associativity) $a \star (b \star c) = (a \star b) \star c$, for all $a, b, c \in G$.
2. (identity element) There is an element $1 \in G$, called the identity, that has the property that $1 \star a = a \star 1 = a$, for all $a \in G$.
3. (inverse element) For each $a \in G$, there exists a value denoted by a^{-1} such that $a \star a^{-1} = a^{-1} \star a = 1$.

An *Abelian group* is a group having the following additional property:

4. (commutativity) $a \star b = b \star a$ for all $a, b \in G$.

For *finite groups* (G finite), the order of a group is defined as the cardinality (size) of G. The *order* of an element a of a finite group G is defined to be the smallest value t such that $a^t := \underbrace{a \star a \star \ldots \star a}_{t} = 1$.

[1] for an introduction to cryptography see [63].

2.1.2 Cyclic Groups

A cyclic group is a group that has the property that there exists an element g such that all elements in G can be expressed as g^i (for different is). If g generates all elements of the group $(G, \star)$, g is a generator and we say it generates $(G, \star)$. Note that the order of a generator g equals the order of the group it generates.

2.1.3 Subgroups

We say that G' is a subgroup of G if $(G', \star)$ forms a group and $(G' \subseteq G)$. If G is a finite group, then the order of a subgroup G' will always divide the order of G (*Lagrange's theorem*, see for example [37]).

2.1.4 Examples of Groups

Groups typically used for DH protocols are the set $\mathbb{Z}_p^*$ with multiplication modulo p where p is prime, the multiplicative group of the field[2] $\mathbb{F}_{2^m}$ and the additive group formed by a collection of points defined by an elliptic curve over a finite field. These groups all have the property that exponentiating is computationally inexpensive and that computing discrete logs is/seems hard (i.e. computationally intractable).

In the remainder of this work, we will take the group to be the set $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$ with multiplication modulo p (p prime) and all operations will be taken over this group (g^y will stand for $g^y \bmod p$, for example). Small variations on many of the attacks of the following sections can be easily mounted on DH implementations using other groups. Note that we will abuse the notation a bit by using $\mathbb{Z}_p^*$ when referring to the group composed of the set $\mathbb{Z}_p^*$ with multiplication modulo p.

2.2 The Core DH Protocol

Alice ($\mathcal{A}$) and Bob ($\mathcal{B}$) first agree on a large prime number p and an element g ($2 \leq g \leq p-2$) that generates a (cyclic) *subgroup* of large order. These values are usually determined a-priori, and are used for many protocol runs (e.g. they could be public parameters that everybody uses). The rest of the protocol goes as follows:

1. $\mathcal{A}$ chooses a number, x, at random from the set $\{1, \ldots, p-2\}$. And $\mathcal{B}$ chooses y randomly from the same set.
2. $\mathcal{A}$ sends g^x to $\mathcal{B}$ and $\mathcal{B}$ sends g^y to $\mathcal{A}$.
3. The shared secret key is $K = g^{xy}$. $\mathcal{A}$, knowing x and g^y, can easily calculate $(g^y)^x = g^{xy}$. $\mathcal{B}$ can determine the secret key in a similar manner by computing $(g^x)^y$.

x and y are referred to as the private keys, g^x and g^y are referred to as the public keys and finally, g^{xy} is called the shared (DH) secret key. Note that an eavesdropper having access to the public keys can't calculate the shared secret key. This protocol is often referred to as *ephemeral* DH secret key agreement because the secret keys are used only once.

2.3 Half-Certified Diffie-Hellman (or Elgamal Key agreement protocol)

This is a very important and useful variant on the Diffie-Hellman protocol discussed above. First introduced in [30], the protocol is almost exactly the same as the basic one except that a user (Bob) publishes his public key (g^y). The public key (g^y) remains constant for large periods of time and is used by everyone wishing to set up a shared secret key with Bob. Note that the public key should be authenticated in some way (e.g. by Bob's signature). This mechanism is especially useful for secure anonymous client connections[3].

[2] see for example [37] for a definition of *field* as well as an overall introduction to algebra.

[3] this scheme is sometimes called Half Static Diffie-Hellman (because one secret, y, is static).

2.4 Attacks

Attacks against the *DH protocol* come in a few flavors:

- **Denial of service Attacks:** Here, the attacker will try to stop Alice and Bob from successfully carrying out the protocol.
- **Outsider Attacks:** The attacker tries to disrupt the protocol (by for example adding, removing, replaying messages) so that he gets some interesting information (i.e. information he couldn't have gotten by just looking at the public keys).
- **Insider Attacks**[4]**:** It is possible that one of the participants in a DH protocol creates a breakable protocol run on purpose (i.e. one in which an outside observer can determine what the shared secret is). Of course, if one of the protocol participants decides to publish the shared secret, nothing can be done. Note that malicious software could be very successful in mounting this attack.

The plausibility of these attacks depends on what assumptions we make about the adversary. For example, if the adversary can remove and replace any message from the public communication channel, the denial of service attack is impossible to prevent. Fortunately, it seems that complete breaks (outsider attacks in which the attacker obtains the shared secret key) and insider attacks can be prevented in many settings[5].

2.5 Man in the Middle Attacks

An active attacker (Oscar), capable of removing and adding messages, can easily break the protocol presented above. By intercepting g^x and g^y and replacing them with $g^{x'}$ and $g^{y'}$ respectively, Oscar ($\mathcal{O}$) can fool Alice and Bob into thinking that they share a secret key. In fact, Alice will think that the secret key is $g^{xy'}$ and Bob will believe that it is $g^{x'y}$. This is a specific instance of a *man in the middle* attack [56].

As an example of what can be done with such an attack, consider the case where Alice and Bob use the "secret" "shared" keys obtained in a DH protocol for symmetric encryption. Suppose Alice sends a message m to Bob and that $ENC_K(x)$ represents the symmetric encryption (e.g. DES) of x using the secret key K.

1. $\mathcal{A}$ sends $ENC_{g^{xy'}}(m)$.
2. $\mathcal{O}$ intercepts $ENC_{g^{xy'}}(m)$ and decrypts it (which he can do since he knows $g^{xy'}$).
3. $\mathcal{O}$ replaces this message with $ENC_{g^{x'y}}(m')$ which he sends to $\mathcal{B}$. Note that m' can be set to any message.

The encryption scheme is thus clearly compromised as message privacy is violated. In the next section, we study attacks that can be mounted by a less powerful adversary.

3 Attacks Based on Number Theory

The previous man in the middle attack, although it completely breaks the protocol, requires Oscar to be very powerful. For example, if the secret keys are used in conjunction with MACs, Oscar needs to intercept and modify each authenticated message in order to prevent Alice and Bob from detecting that their keys aren't identical. In some of the following subsections, Alice and Bob have the same secret key (which Oscar knows). Thus, Oscar only needs to be active during the DH protocol, afterwards he can break the protocols using the shared secret key whenever he wants.

[4] these are sometimes referred to as Byzantine errors.

[5] in practice it is much easier to insert packets than it is to delete them. In any case, we consider all attacks in order to derive a DH protocol that is secure in all practical settings.

3.1 Degenerate Message Attacks

There are degenerate cases in which the protocol doesn't work (i.e. it can be broken). For example when g^x or g^y equals one, the shared secret key becomes 1. Since the communication channel is public anybody can detect this anomaly. Fortunately, this situation is impossible in a properly carried out protocol run because both x and y are chosen from $\{1, \ldots, p-2\}$ [6]. However, an insider attack is possible and so DH protocol participants should make sure that their key agreement peer doesn't send $g^z = 1$.

3.1.1 Simple Exponents

If one of x and y can be easily determined, the protocol can be broken. For example, if x equals 1 then $g^x = g$ which any observant attacker will be able to detect. It's very hard to determine where to draw the line here, that is, determining for which values of g^i, i is hard to determine. In any case, it seems very reasonable to insist that x and y not equal 1. Another option is to insist that the secrets x and y have length[7] at least l (i.e. $x, y \geq 2^{l-1}$); this approach is found in RSA security's PKCS#3 standard for example.

3.1.2 Simple Substitution Attacks

The following attack is very interesting, as it is extremely easy to mount and normally wouldn't come up in theoretical proofs of security. The attacker can force the secret key to be an "impossible" value. If the DH protocol would only be executed by sentient beings this wouldn't be interesting as the anomalies would be easily detected. However in practice DH protocols are carried out by computers and careless implementations might not spot the following attack.

1. $\mathcal{O}$ intercepts g^x and g^y and replaces them with 1.
2. Both $\mathcal{A}$ and $\mathcal{B}$ compute the same shared secret key which equals one.

If the computer program doesn't realize that g^x, g^y and g^{xy} can't equal 1, the protocol is vulnerable. Note that the same argument holds for values of the form $g^{\alpha \cdot (p-1) \cdot x}$ or $g^{\alpha \cdot (p-1) \cdot y}$, where $\alpha \geq 1$, because a computer might not realize that these values have not been computed modulo p. (They equal 1 modulo p). So it is safe practice to always verify that in fact g^x, g^y is a positive integer smaller than $p-1$.

The following attacks delve a bit deeper into computational number theory.

3.2 Generators of Arbitrary Order and the Pohlig-Hellman Algorithm

The Pohlig-Hellman algorithm [53] allows one to efficiently compute the discrete log of g^x if the prime factorization of g's order consists of small primes. Precisely, given that the order of a group has the following prime factorization, $p_1^{e_1} p_2^{e_2} \cdot \ldots \cdot p_r^{e_r}$, the Pohlig-Hellman algorithm's computational complexity is $O(\sum_{i=1}^{r} e_i(lg(n) + \sqrt{p_i}))$. A secure DH implementation must make this algorithm impractical. A simple solution is to choose a prime p such that $p-1$ contains large factors. Safe primes, primes of the form $p = Rq + 1$ (where R is some small positive value and q is a large prime[8]), and Lim-Lee primes [45] which have the form $p = 2q_1 \cdot \ldots \cdot q_n + 1$ (where the q_is are all large primes) satisfy this property. In these last cases, prime factorization of the order of each generator will contain a large prime which is exactly what we need to make this attack impractical. (Remember that the order of any subgroup will divide $p-1$, i.e. the order of $\mathbb{Z}_p^*$.)

3.3 Attacks Based on Composite Order Subgroups

The attacker can exploit subgroups that do not have large prime order [64]. This is best illustrated by an example. Suppose Alice and Bob choose a prime $p = 2q+1$, where q is prime, and a generator g of order $p-1 = 2q$. Oscar can intercept the messages g^x and g^y and exponentiate them by q. (He will replace g^x by g^{xq} and g^y by g^{yq}.) The secret key will be g^{xyq} which allows Oscar to find this value by exhaustive search. This is done by noting that the order of

[6] if g is a generator of $\mathbb{Z}_p^*$, $g^z = 1 \bmod p$ iff $z = 0 \bmod p-1$.

[7] the length of an integer n is defined as the largest value k satisfying $n \geq 2^k$.

[8] a prime q such that $2q+1$ is also prime is called a Sophie Germain prime.

$g^q = g^{\frac{p-1}{2}}$ is[9] 2 which implies that the secret key can only take two values! Hence, Oscar can use a brute force search (only two elements to try) in order to determine what the shared secret key is; for example, when Alice and Bob use it for symmetric encryption.

More generally, this attack can easily be mounted on primes of the form $p = Rq + 1$ (R small), the only difference being that there are R possible values to try in the exhaustive search.

The lesson to be learned from this attack is that we should choose a g that generates a large prime order subgroup or at the very least make sure that composite order subgroups aren't vulnerable (e.g. the order's prime number factorization contains only large primes). Note that an attack of this type is part of the motivation for using DSA instead of Elgamal signatures. In essence DSA is an immunized version of Elgamal [6].

Notice that an insider attack can be mounted using this trick. Alice simply chooses x to equal q. In this case, even authentication mechanisms can't protect Bob.

3.4 Pollard Lambda Algorithm

The Pollard Lambda method [54] enables one to compute z given g^z, when z is known to be in a certain interval $[b, b + w]$ in time $O(w^{1/2})$. This is an extremely relevant attack to consider when we want to limit the exponent range to improve efficiency. For example, when $x, y < 2^N \ll p$ the attacker can compute x and y (given g^x and g^y) in $O(\sqrt{2^N}) = O(2^{N/2})$. Hence, if we want the attacker to execute at least (2^N) operations[10], x and y need to have a length of at least $2N$ bits. We note that this attack hasn't been improved in a long time and many cryptographers feel that it's improbable that the state of the art for this kind of attack will change (this is a useful observation when choosing key sizes). Also remark that this attack can be mounted on subgroups of small order.

3.5 The Number Field Sieve Algorithm

It is obviously important to choose a group (i.e. p) large enough so that the best known algorithms for computing discrete logs are intractable. The state of the art, index calculus based, methods for computing discrete logs (number field sieves) have been steadily improving[11] over the years and so it's harder to gauge how large p should be for long term security. In [49], Odlyzko proposes using a p of at least 1024 bits for moderate security and at least 2048 for anything that should remain secure for a decade.

3.6 Attacks on Prime Order Subgroups

In [45], an attack on prime order subgroups is presented (a slight extension of the ideas of [64]). The attack can be mounted if the protocol doesn't satisfy the sixth robustness principle of [5] which states:

> *Do not assume that a message you receive has a particular form unless you can check this.*

The idea is that if we can get a participant with secret key x to use an arbitrary group element instead of g^y then we may be able to obtain some information about x. If the attacker can obtain γ^x, for some generator γ whose order's prime factorization contains only small primes, then he can use the Pohlig-Hellman algorithm of subsection 3.2 to obtain x modulo the order of γ.

To obtain the actual value of the secret key (modulo p), a slight variation on the Pollard lambda method [64] might be feasible.

The problem with the above situation is that the attacker still needs to obtain γ^x, which isn't obvious. Fortunately, Lim and Lee [45] give a weaker version of the previous attack that enables the attacker to obtain the value of x modulo the order of γ in time linear in the order of γ.

This method can be easily foiled if the participants check that the value they receive (i.e. usually g^z) has order q. This can be done by verifying that exponentiating the value by q yields 1. If p is of the form $p = 2q + 1$, with q prime, the best one can hope for is to determine the parity of the secret key.

If we have Certificate Authorities (CAs) certify the DH public keys, they must be wary of this attack. It is usually sufficient for the CA to verify that the user knows the secret associated with the public key. This is usually done by

[9] the subgroup generated by $g^{\frac{p-1}{2}}$ is $\{g^{\frac{p-1}{2}} = p - 1, (g^{\frac{p-1}{2}})^2 = 1\}$.

[10] Assuming that the Pollard Lambda technique is the best method in this situation.

[11] As opposed to the Pollard Lambda type algorithms for which there hasn't been substantial progress for about twenty five years [49].

having the user sign some message with the secret key. Unfortunately, a variation on the previous attack allows for an insider attack where a user can fool the CA when specific signature schemes are used (e.g. Schnorr signatures [57], see [45] for the details). Hence, when this type of attack can be mounted, we should check the order of the public keys.

4 Authentication

In the previous section we presented attacks related to the mathematical structure of the DH protocol primitives. In this section we address issues related to authentication. As the DH protocol can be broken by a simple man in the middle attack (if no authentication mechanism is used), it doesn't make sense to talk about DH protocol security without also discussing authentication.

Authentication consists of establishing authenticity, which is defined as: *factually accurate and reliable*. This is a somewhat slippery concept and there are no solid and formal definition, because the different settings and requirements change for every application. For example, validating the authenticity of a digital signature or a MAC is simple (just apply a verification function) whereas proving the authenticity of a message is more complicated. For example, if Alice sends a message to Bob, he might want to:

1. Establish that the message hasn't been modified.
2. Establish that Alice sent the message.
3. Establish that the message was meant for him (i.e. addressed to him).
4. Establish that the message hasn't been "replayed".
5. Establish that the message was sent within a certain time period.

Although we have some very powerful primitives that can help us in creating authentication mechanisms (digital signatures, MACs, symmetric encryption, etc.), using them in an effective manner is surprisingly difficult.

4.1 Message Replay Attacks

One of the deadliest attacks against authentication mechanisms is the message replay attack [48] in which the adversary simply takes a previously sent message and sends it again. This attack is deceptively powerful as can be seen by the next example: suppose a user sent a message to his wife saying, "I love you". A few years later, after the user has been divorced, an attacker could re-send this same message which might lead to an awkward situation. If a correct authentication mechanism is used, the now ex-wife will not consider the message as being authentic. This example nicely illustrates the fact that digital signatures aren't sufficient to establish message authenticity.

4.2 Message Redirection

If the destination isn't specified in a message, an attacker can intercept it and send it to someone other than the intended recipient. Taking the previous subsection example's premise, the adversary sends the "I love you" message and delivers it to somebody other than the intended recipient; which again, might lead to an uncomfortable situation.

The two previous schemes form the basis of many other, more involved, attacks.

4.3 Message Authentication Protocols

We now present one of the authentication mechanisms described in [9]. Note that the protocol is proved to be secure (see subsection 1.1). It has the property that if we have a scheme that is provably secure when the channels are authenticated, and replace the sending mechanism with the following protocol, then the resulting scheme will be provably secure in a setting in which the channels aren't authenticated.

In the following protocol, we assume that the users' public keys are certified by a certificate authority (CA), and by authentic we mean:

- The sender's identity is established

- The intended recipient's identity is established
- Context is established; message replay attacks such as the one in the example in subsection 4.1 are prevented.

Alice sends a message m to Bob, who establishes its authenticity.

1. $\mathcal{A}$ sends m to $\mathcal{B}$.
2. $\mathcal{B}$ replies with a challenge $N_{\mathcal{B}}$ and m, where $N_{\mathcal{B}}$ is a random number (nonce). Note that each n*once* is only used *once*.
3. $\mathcal{A}$ sends m and $SIG_{s_{\mathcal{A}}}(m, N_{\mathcal{B}}, \mathcal{B})$, where $SIG_{s_{\mathcal{A}}}(x)$ is a digital signature on x that uses $\mathcal{A}$'s secret key $s_{\mathcal{A}}$.
4. If the signature verification procedure is successful then m is deemed authentic.

Let's look at this protocol a bit more closely and explain some of its features and propose some efficiency improvements.

First note that sender authenticity is established by the public key that is used to verify the signature. The public key and Alice's identity are certified by a CA that Bob trusts.

The message m must, of course, be sent at some point. The protocol is still provably secure if the message is sent in one of the first or third rounds. For example, the first message could simply be a synchronization signal.

The need for a nonce $N_{\mathcal{B}}$ is quite interesting. Notice that without it, the protocol would be vulnerable to replay attacks. By slightly modifying the protocol we can, however, do without the nonce. If Bob takes note of all the messages he has received, and so can detect message replay attacks, we can omit the nonce. Unfortunately the amount of data Bob would need to keep could be huge. Also, deterministic signature schemes (e.g. RSA) are not well suited to this setting as two signatures on the same message are identical. (As opposed to probabilistic encryption schemes such as Elgamal.) Note that random numbers can be appended to the message in order to make signatures on the same message different. Another option is to have publicly available nonces. For example a counter can be used, in which case Bob needs to manage counters (synchronization is often difficult to implement). As long as nonces (counter values) are only used once the protocol is correct. In both of these modifications, the second round of communication can be omitted and the first and third rounds combined.

The last interesting issue to point out is that Alice must sign $\mathcal{B}$, i.e. Bob's ID. If this isn't done, the protocol is vulnerable to message redirection attacks.

Note that the CA *must* verify that the client knows the secret key associated with his public key, otherwise the protocol is vulnerable to message hijacking (i.e. claiming ownership of someone else's message). Note that self signed certificates[12] can also be used to solve this problem as is done in PGP [52].

5 Attacks on Implementation Details

5.1 Attacks on Parameter Authentication

As a general principle, all parameters used in a cryptographic protocol should be authenticated. For example, suppose that the DH protocol could be used with different system parameters (e.g. g, p); if the participants do not authenticate their choice of parameters, an attacker might be able to fool them into using weak parameters. These types of attacks can be very subtle and can even be missed by top cryptographers and security experts. One need just look at the attack of [65] on the SSL protocol version 2 to be convinced of this[13].

5.2 Context

In many situations it is necessary to make sure an adversary hasn't blocked (deleted) previous messages. This can be done by simply hashing all previous messages and appending the result with the current message. This establishes *context*. Note that if *all* messages are authenticated we could use a sequence number, this would be much more efficient.

[12] certificates that are signed by the subject of the certificate.

[13] SSL version 2 was vulnerable to what is called a version roll-back attack which is an attack on parameter authentication, see [65] for details.

5.3 Race conditions

In most, if not all, networking protocols, it is very important to preserve protocol run independence and avoid race conditions. That is, we don't want messages used in one protocol run to be used by another protocol execution. Session numbers, for example, can be used to prevent this kind of problem.

The first Freedom[14] DH implementation didn't respect this design principle. In this particular instance, if two parties initiated a DH protocol at the same time, each party obtained two shared DH secret keys and it was possible to have a situation in which none of the (four) supposedly "shared" DH secret keys were equal.

A key agreement confirmation (see 6.5) is a way of making sure problems such as the one described above don't occur.

5.4 Deleting the ephemeral secrets

It is important to delete the ephemeral secret keys (the secret exponents), to guard against memory being written to disk (swapping) and the possibility that unauthorized entities might have access to these values. Deleting private values is usually done by overwriting these values with some constant (by 0s for example). It is important that the values be deleted as soon as possible to guard against RAM reading techniques such as the ones described in [34].

5.5 Bleichenbacher Type of Attacks

D. Bleichenbacher described in [13] an attack against PKCS #1 v1.5. The attack exploited the fact that some servers implementations of the PKCS #1 v1.5 RSA encryption padding used an inadequate authentication mechanism: if a plaintext started with 0002, as described in the standard, they would blindly accept it as valid and continue, otherwise, they would return an error message to the client. Using a theorem due to Chor [22], Bleichenbacher devised a practical attack against some implementations of SSL v3.0.

Although we don't describe any padding methods, nor do we use the RSA encryption scheme, some of the proposed countermeasures (see [14]) to immunize protocols against this attack are relevant:

- Change keys frequently (as discussed in section 6.2) and make sure that different servers use independent keys.
- Use only adequate authentication (as discussed in section 4.3). Servers written in SSL version 3 that used adequate authentication weren't vulnerable to this attack.

5.6 Timing Attacks

An interesting attack was proposed in [42]; the attack relies on the fact that for most modular exponentiation algorithms the time taken is dependent on the inputs. In the Half Certified DH protocol, an attacker, by initiating many protocol runs with Alice and carefully choosing his "public keys", could determine Alice's secret key (x). Remember that Alice computes m^x in each protocol run (where m can be the attacker's "public key" (simply a random value) and x is Alice's secret key). Fortunately, the attack is only effective if the attacker can somewhat precisely determine Alice's computing time. The attack can be countered by modifying the computations so that the exponentiation time doesn't depend as heavily on the input parameters.

Kocher [42] gives a method that uses the *blinding* techniques of [21] that help randomize the modular exponentiation computing time[15] : (we provide pseudo-code in section 8.1.4.)

One Time Set Up:
We calculate the *private* seeds.

- An integer, v_i^0, is chosen at random from $\mathbb{Z}_p^*$.
- $v_f^0 = ((v_i^0)^{-1})^x$ is calculated. (Find inverse of v_i^0 and exponentiate by x.)

[14] www.freedom.net.
[15] thus blinding the attacker from information about x.

j'th Exponentiation:
Let m be the message to be exponentiated by x.

- Calculate $u = (m \cdot v_i^j)$. (the blinding part.)
- Calculate $t = u^x$. (u is not known to the attacker!)
- Calculate $t \cdot v_f^j$ which is equal to m^x. (the unblinding part.)

Computing the j'th seeds (j > 0):

- $v_i^j = (v_i^{j-1})^2$.
- $v_f^j = (v_f^{j-1})^2$.

The technique uses the fact that if v_i^0 is chosen randomly, then the series $(v_i^0, v_i^1, \ldots, v_i^j, \ldots)$ will have the property that v_i^j looks sufficiently random if nothing is known about the previous elements of the series[16]. The algorithm has to keep in an internal state the most recent v_i and v_f. These values can be kept in some sort of structure, what is important is that they must remain secret!

5.7 Denial of Service Attacks (Overloading)

One of the most effective attacks in practice consists of overloading servers with requests[17]. This is a type of denial of service attack because the server is so busy processing bogus requests that he doesn't have time to reply to legitimate queries. The adversary usually exploits the fact that the servers are limited in terms of memory [20, 23] and/or computational power. The DH protocol is vulnerable to the following kinds of attack:

- The attacker can carry out a connection (memory) depletion attack (e.g. [20, 23]). Note that it is very important that the low level protocols for sending and receiving messages be immunized against this attack.
- The attacker can send huge amounts of public keys (which can simply be random numbers) so that the victim is compelled to carry out many modular exponentiations in order to compute the shared DH secret keys (computational).

The most robust solutions [7, 29, 38] to the problem involve having the connection initiators compute a solution to cryptographic puzzles (also known as hashcash or pricing functions). The amount of computations needed to solve these puzzles is small enough so that legitimate users can quickly compute the solution but large enough so that it's infeasible (or at least very hard) to solve a large number of them for use in overloading attacks.

If a server can validate the IP addresses of it's clients, one can use a less robust scheme for protecting against denial of services called SYN Cookies ([46], [16], [39]). SYN Cookies help prevent IP spoofing to a certain extent.

If a server is suppose to be able to accept unknown clients (or better yet anonymous clients), we suggest using the techniques of [38] which we now present and discuss. Note that $x_{<a,b>}$ refers to the substring consisting of the a'th through b'th bits of x. $\mathcal{C}$ is the client and $\mathcal{S}$ is the server.

1. $\mathcal{C}$ requests a puzzle from the server (stateless connection).
2. $\mathcal{S}$ sends $x = H(s, t, k, m, C)$, as well as m (the number of sub-puzzles), k (a computation parameter) and t (a timestamp). Note that $H()$ is a cryptographic hash function, s is $\mathcal{S}$'s secret and C is $\mathcal{C}$'s address.
3. For i equals 1 to m, $\mathcal{C}$ finds z_i such that the first k bits of $x \| i \| z_i$ equal the first k bits of $H(x \| i \| z_i)$, where $\|$ denotes concatenation. The z_is (i.e. the solutions to the sub-problems), t and $\mathcal{C}$ are sent to $\mathcal{S}$.
4. $\mathcal{S}$ receives these values and can efficiently check that the solutions are valid and that they have been computed in a timely manner.

The value of C is usually implicitly determined, for example it might be included in the message headers. The server sends his replies to C and so the adversary must be able to intercept messages addressed to C which is difficult if the adversary isn't located at C[18].

[16] this is much more efficient since modular squaring is a lot cheaper than choosing a new random value.
[17] see for example http://www.cisco.com/warp/public/707/newsflash.html.
[18] this prevents straightforward IP spoofing; this alone is also achieved by SYN Cookies ([46], citeRFC1644, [39]).

If no time parameters are used, an attacker could obtain a large number of puzzles, solve them (which can take a lot of time) and then overload the server. By encoding t in x using the secret s, the puzzles can be made to have a limited validity period which makes the previous attack infeasible (all ts should be different). (Note also that if connections have an unlimited lifetime, the server is vulnerable to denial of service attacks and so maximum connection lifetime must be taken into consideration when choosing our parameters.)

Attacks are a rare occurrence and so it makes sense to be flexible in our use of puzzles. Precisely, we should vary k depending on the situation: The busier the server is, the larger k should be (we can omit puzzles altogether in most situations).

s should be large enough so that it can't be obtained by a brute force attack.

See section **??** for a discussion on parameter sizes and lower level issues.

6 The DH Shared Secret Key

The shared secret obtained is usually used to derive session keys that will be used in other applications. Now, the operations these keys will be used for have their own requirements and security vulnerabilities. If we are not careful in how we use the shared DH key, we might be vulnerable to other subtle attacks.

6.1 Key Derivation Function (KDF)

In most, if not all, instances we need to modify the shared secret key obtained in the DH protocol in order to use it with other cryptographic primitives. Here are the main motivations for "modifying" the shared DH secret key:

- The key sizes might not correspond. For example suppose we want to use our a-bit shared secret DH key with a crypto-system requiring a key size of b, with $b \neq a$.
- Although some bits of the shared secret are provably secure [15] the security of the vast majority of bits in the shared DH secret key is not known (i.e. it's not known whether an attacker can compute information about them[19]).

  Also notice that $\mathbb{Z}_p^*$ doesn't span all the bit-strings of length p (for example, we will never get the value $p+1$). Hence if we take a random number, chances are greater that the most significant bit equals 0.

  Hence, it makes sense to spread the risk and have the bits in the new session key depend on *all* the bits of the shared DH secret key.
- Some attacks exploit algebraic relationships between keys (see section 6.3). Hence, it is important to destroy mathematical structure which can be done using a KDF.
- If we want to create more than one session key with a given shared secret DH key then, if the KDF is a carefully chosen one-way pseudo random number generator, the system can be resistant to known session key attacks (i.e. given a session key, it's hard to find other session keys derived using the same shared secret DH key).

We propose two key derivation function constructions (HS is a secure cryptographic hash function and $||$ denotes concatenation):

1. **The Counter Based Approach:**
   We take the bits for the session key from:

   $$\text{HS(shared DH secret } || \text{ 0) } || \text{ HS(shared DH secret } || \text{ 1) } || \ \ldots \ || \text{HS(shared DH secret } || \text{ c).}$$

   c's value will depend on the number of bits required. This method is similar to the ones that are used in many standards (e.g. TLS [26]).
2. **A Chaining Based Approach:**
   We take the bits for the session key from:

[19] in fact, in some cases it is trivial to compute information about the shared secret g^{xy}, given only p, g, g^x and g^y.

s_1 = H(shared DH secret)
s_2 = H(s_1|| shared DH secret)
...
s_c = H(s_{c-1}|| shared DH secret)

c's value will depend on the number of bits required.

This last approach however isn't as efficient and can't be parallelized.

6.2 Key Freshness and Perfect Forward Secrecy

In many situations the shared DH secret key should be changed frequently. Here are the main reasons why we might want to obtain new shared secret keys often.

1. **Reduce Exposure:**
   The probability that a given key is compromised is lower if it isn't used often.
2. **Damage limitation :**
   If the amount of traffic encrypted/authenticated with a given key is reduced then the amount of damage done if the key is compromised is reduced.
3. **Forward Secrecy:**
   If old encryption keys are deleted, encrypted messages can no longer be decrypted. Hence, a third party can't mount a subpoena attack (i.e. demand that old messages be decrypted).

As expected, tricks used to improve the efficiency of schemes in which keys are changed often have subtle problems (see subsection 6.4).

6.3 Key Independence

As a general principle, we always want keys to be independent. Precisely, obtaining one secret key should *not* help an attacker uncover other keys. This property is called *known key security*. In the next subsection, we will give an example of a protocol vulnerable to known key attacks.

6.4 An Example

We now present a condensed version of the KEA protocol [3] which is a part of the NSA's FORTEZZA suite of cryptographic algorithms and motivate the use of key derivation functions. Note that the explanations roughly follow those of [12].

1. $\mathcal{A}$ gets $\mathcal{B}$'s static public key g^y sand $\mathcal{B}$ gets $\mathcal{B}$'s static public key g^x (x is secret to $\mathcal{A}$ and y is secret to $\mathcal{B}$). (respectively) certified by a CA. (x is Alice's private key and y is Bob's private key.)
2. $\mathcal{A}$ sends g^a, g^x and Cert($\mathcal{A}, g^x$) to $\mathcal{B}$ and $\mathcal{B}$ sends g^b, g^y and Cert($\mathcal{B}, g^y$) to $\mathcal{A}$. (a and b chosen randomly from the set $\{2, \ldots, p-1\}$) Note that Cert(x) is just a certificate certifying x.
3. If all verifications succeed, the shared DH secret key is taken to be $K = g^{ay} + g^{bx}$.
4. A key derivation function (derived from SKIPJACK) is then applied to K to obtain the key (the session key) that will be used in the other applications (e.g. encryption, MAC, etc.).

The protocol solves many of the problems mentioned in the previous subsections:

- **Key Freshness:** We can obtain as many fresh keys as we need without having the CA re-certify new public keys every time.
- **Forward Secrecy:** If Alice and Bob delete K and *both* the static and ephemeral secret keys (x and a respectively for Alice) we have forward secrecy.

- **Key Independence:** The protocol *seems* resistant to known key attacks.
- **Key Derivation Function:** The session key depends on all of the bits of the shared DH secret key. As will be seen shortly, the key derivation function is also important because it destroys the algebraic relationships between keys.

If the protocol didn't use a key derivation function, it would be vulnerable to the Burmester triangle attack [17] which renders the protocol vulnerable to known key attacks. In the previous protocol, if a key derivation function isn't used, it is vulnerable to the following attack:

1. $\mathcal{O}$ first observes a protocol run between $\mathcal{A}$ and $\mathcal{B}$. He obtains the ephemeral keys g^a and g^b. The key shared by $\mathcal{A}$ and $\mathcal{B}$ at the end of the protocol is $K_{\mathcal{AB}} = g^{ay} + g^{bx}$.
2. $\mathcal{O}$ then engages $\mathcal{A}$ in a protocol run. $\mathcal{O}$ will use g^b as his ephemeral key and g^z as his static key. Assuming $\mathcal{A}$'s ephemeral key is $g^{\overline{a}}$, the shared key will equal $K_{\mathcal{AO}} = g^{bx} + g^{z\overline{a}}$.
3. $\mathcal{O}$ carries out the same trick with $\mathcal{B}$ but now uses g^a as his ephemeral key. Assuming that $\mathcal{B}$'s ephemeral key is $g^{\overline{b}}$, the shared key will be $K_{BO} = g^{ay} + g^{z\overline{b}}$.
4. *If* $\mathcal{O}$ can obtain $K_{\mathcal{AO}}$ and $K_{\mathcal{BO}}$ he can determine $K_{\mathcal{AB}}$. This can be seen by noting that $K_{\mathcal{AB}} = K_{\mathcal{AO}} + K_{\mathcal{BO}} - g^{\overline{b}z} - g^{z\overline{a}}$.

6.5 Key Agreement Confirmation

In some settings, the participants won't settle with just knowing that nobody except the intended party can compute the session key (i.e. a key derived from a shared secret DH key) but insist on having some kind of confirmation that a secret key has been (or can be) successfully created. A scheme provides *i*mplicate key confirmation if the participants can be convinced that they all *can* compute a common shared secret key, and provides *e*xplicit key confirmation if participants can be assured that a common shared secret key *has* been computed by all participants. The simple minded solution to providing explicit key agreement is to have the parties compute the MAC (using the new session key) of a known message. Unfortunately this means that the key will be distinguishable from a random key (we know the MAC of a known message). If indistinguishability is required we need to use (as a MAC key) some other value, r, known only to the participants that can't be easily linked to the session key. Precisely, given the session key it should be computationally infeasible to find r. See for example [12] for techniques that can be used.
Although explicit key confirmation appears to provide stronger assurances, implicit key confirmation is sufficient in practice. Also, it would seem that although it is possible to provide explicit confirmation of the derived shared secret key without using any previous shared secret, it is impossible to provide explicit confirmation of the DH shared secret without a previously shared secret, so it's usefulness is questionable.

7 The Bottom Line

In this section, we give recommendations that are, for the most part, based on the lessons learned in the previous sections. These can be seen as general robustness principles[20] that should be taken into account when implementing DH key agreement type protocols.

Note that DH protocol implementations should take into account attacks that allow the attacker to obtain static secret keys (e.g. x, y). Compromising static secret keys allows the attacker to break all subsequent protocols using these values. Compromising g^{xy} doesn't help in compromising other shared DH secret keys $g^{xy'}$.

7.1 Diffie-Hellman Math

1. **Spot Unconventional Messages**
   - Make sure that g^x, g^y and g^{xy} do not equal 1.

[20] a DH version of [5].

- Make sure that g^x and g^y are less than $p-1$ and greater than 1.
- Choose x, y, from the set $\{2, \ldots, p-2\}$.

2. **Be Careful About g's Order**
   - The prime factor decomposition of the order of the g's shouldn't be composed entirely of small primes.
   - The subgroup generated by g should not have a small order subgroup. If at all possible, construct and use a generator that has a large prime order.
3. **Make Sure the DH Public Keys Received Have the Correct Order**
   - The DH public key's (g^x) order should be checked. This can be easily done by verifying that $(g^y)^{\overline{o}} = 1$ where $\overline{o}$ is g's order. If $p = 2q + 1$ this isn't necessary as explained in section 3.6.
4. **Make Sure the System Parameters Aren't Chosen Maliciously**
   - The system parameter's properties should be known (subgroup generated by g's order, prime factorization of this number, etc).
   - Proofs that the parameters have been chosen at random should be available. This can be done by kosherizing (see section 8.2.6).
5. **Choose Secure Parameters**
   - Cryptographic algorithms are only as secure as their weakest link and so it makes sense to try and balance the security. That is, attacks that exploit different parameters of the system should take roughly the same amount of time. For the DH protocol, the parameters to balance are : the value of p, the exponents' range and the size of the keys derived from the shared DH secret. We suggest looking at [44] for a table of balanced values.
   - The parameters should be chosen in order to provide good long term security. Note that parameters that constitute "good" long term security is very controversial [44, 59]. Extremely conservative estimates are (from [44]):
     - For very good security until 2002 take: p 1024 bits, exponent range 127 bits and derived key length 72.
     - For very good security until 2025 take: p 2174 bits, exponent range 158 bits and derived key length 89.
     - For very good security until 2050 take: p 4047 bits, exponent range 193 bits and derived key length 109.
     - We suggest using strong primes or Lim Lee primes so as to guard against the attacks presented in section 3.5.

     Note that these values are very controversial [59], the size of p is especially debatable since it assumes Moore's law type improvements in algorithmic number theory.
   - The number of symmetric keys derived from the shared DH secret key should also be taken into consideration when determining the size of p and of the exponent range since breaking the DH protocol breaks *all* derived keys. Precisely, if we derive n session keys of lengths $n_1, n_2, \ldots, n_k$, our other parameters should, *in theory* provide the same security as if we derived one session key of length $lg(2^{n_1} + 2^{n_2} + \ldots + 2^{n_k})$.

These precautionary measures protect against some insider attacks and man in the middle attacks.

7.1.1 Efficiency Considerations

1. Ideally, the generator g should be as small as possible in order to reduce the cost of modular exponentiation. Wiener and van Oorschot [64] claim that using $g = 2$ reduces the computation time for modular exponentiation by 20% (compared to randomly selected generators). For applications in which efficiency is crucial and the prime numbers can be generated beforehand, it makes sense to find a prime, p, such that a small value (e.g. 2,3,16) generates the desired subgroup.

2. Generating safe primes ($p = 2q+1$) is more expensive than generating Lim-Lee primes ($p = 2q_1q_2 \ldots q_n+1$). If very efficiently generating the parameters is important, we suggest the use of Lim-Lee primes. If the parameters are fixed, we suggest the use of Sophie Germain primes since they enable the use of larger exponents (thus resulting in larger shared secrets). Note that there exist particular primes that yield more efficient operations, see section 8.2.6.

3. Exponentiations are usually much faster when the exponents are small and so we suggest using the smallest secure exponent range (see subsection 7.1).

7.2 Implementation Details

Correctly establishing authenticity is difficult and when possible, provably secure authentication protocols should be used (at the very least, the attacks mentioned previously must be taken into account). Particular care must be taken when improving a protocol's efficiency (e.g. removing "superfluous" messages).

Note the following tricky implementation level issues:

1. **Exact Destination:** The message recipient should be precisely specified. Identification fields could include IP address, port number, user ID, etc.

2. **Multiple Session Management:** It is of crucial importance for participants to separate concurrent protocol executions. Concurrent protocol executions should be *independent*. This problem can usually be dealt with by adding a session ID field to the messages. Note that this is a tricky problem to solve when sessions are related in some way, for example when counters are used instead of nonces.

3. **Certifying Public Keys:** The certifying authority should, of course, be trusted by both participants. As pointed out earlier the certificate authority should make sure that the certificate recipient knows the secret key associated with the public key being certified. Another option is to have the sender sign his identifier along with the rest of the message (i.e. self signed certificates).

4. **Authenticate Parameters:** *All* parameters should be authenticated.

5. **Previous Communications:** In many situations, all messages sent should "confirm" all previous messages. We want to avoid some messages being blocked. This can be done by appending a hash of all previous messages or by using sequence numbers.

6. **Delete Useless Key Material:** When some sensitive information is no longer needed it should be securely deleted.

7. **Timing Attacks:** When a system is vulnerable to timing attacks (see section 5.6), a special exponentiation routine should be used.

8. **Denial of Service Attacks (Overloading):** When necessary (see section 5.7), parties should protect themselves against denial of service attacks (overloading).

7.3 Using the Shared DH Secret Key

Here are the general points related to the utilization of the shared DH secret key.

1. **Never use the key "as is":** Always use a *suitable* key derivation function in order to get a session key.

2. **Key Independence:** It is important for the protocols to be resistant to known key attacks.

3. **Delete Old Keys:** If forward secrecy is desired old keys and all data that can be used to obtain them must be securely deleted.

4. **Be Careful with Confirmation:** If key indistinguishability is desired, we can't just send the MAC of a known message.

8 Pseudo-code

In this section we give fairly detailed pseudo-code which should bridge the gap between programmers and mathematicians. That is, a mathematician can painlessly check that the protocol is correct while a programmer can easily understand what needs to be done.

We present useful mathematical primitives in subsection 8.1, cryptographic primitives in subsection 8.2, high level data structures related to DH in subsection 8.3, the send and receive primitives in subsection 8.4, an authenticated ephemeral shared DH key protocol in subsection 8.5, client puzzles in subsection **??**, connection primitives resistant to denial of service attacks in subsection **??** and finally a half-certified DH protocol in subsection **??**.

8.1 Mathematical Primitives

The following mathematical primitives are needed in the following protocols. See for example [47] for more details on the algorithms and [58] for free librairies that implement them.

8.1.1 Modular Multiplication

mult(a,b,p);

Returns $a \cdot b \bmod p$.

8.1.2 Modular Squaring

square(a,p);

Returns $a^2 \bmod p$.

8.1.3 Inverting function

inv(a,p);

Returns a's inverse in $\mathbb{Z}_p^*$, that is an element a^{-1} such that $a^{-1} \cdot a = 1$.

8.1.4 Modular Exponentiation

mod_exp(g,x,p);

Returns $g^x \bmod p$.

Since this operation might be vulnerable to the timing attacks of subsection 5.6 (such is the case in the Half-Certified DH), we define an exponentiation algorithm resistant to timing attacks : blind_mod_exp(g,x,p);

Pseudo-Code

One Time Set Up:

```
/*
 *   Goal: Compute initial values for vi and vf that will be
 *         used in blinding exponentiation.
 *   Given:
 *     Secret Exponent -> x
 *     parameter -> p     working in Z_p
 */

vi = PRNG(2,p-1);  //choose a random number from {2,...,p-1}

/*
 * Find vi's inverse and exponentiate by x
```

```
 * (ordinary exponentiation -- not a recursive call)
 *
 */

vf = mod_exp(inv(vi),x,p);
```

Exponentiation

```
/*
 *   Goal:
 *      Secure Exponentiation
 *   Given:
 *     Secret exponent -> x
 *     parameter -> p
 *     Peer's public key -> m  (base)
 *     Current Seeds -> vi and vf
 */


/***** This can be pre-computed *****/
vi = sq(vi,p);   //The square of the previous seed
vf = sq(vf,p);   //The square of the previous seed
/***********************************/

temp = mult(vi, m, p);
temp = mod_exp(temp, x, p);  //typical exp (not a recursive function call)
temp = mult(temp, vf, p);

return temp;  //remove all traces of temp (e.g. overwrite with zeroes)
```

Comments: The main disadvantage of this function is that it needs an internal state (vi and vf) which has to be kept secret. Note that the blinding (vi) and unblinding (vf) factors can be pre-computed.

8.2 Cryptographic Primitives

We refer the interested reader to [47] for an in depth analysis of many of the following primitives.

8.2.1 Cryptographic hash functions

Denoted by HS(), HW();

SHA1 [61] and RIPEMD-160 [28] (the output is 160 bits long) are thought to be the most secure hash functions. MD5 [55], although it hasn't been broken, has become a more dubious choice since the discovery of internal pseudo-collisions ([25]). MD5 (or MD4 which is faster and has weaker security) might be useful in protocols that don't have stringent security requirements because it produces a 128 bit output and is much faster than both SHA1 and RIPEMD-160.

HS() refers to strong hash functions (SHA1-160, RIPEMD-160, etc.) and HW() refers to weak hash functions (SHA1-80, MD5, MD4, etc.) which should only be used for compression and situations in which a security/performance tradeoff makes sense. (Note that SHA1-80 isn't as efficient as MD5 which isn't as efficient as MD4).

8.2.2 Message Authentication codes MAC

Denoted by $\mathsf{MAC}_K()$ (K is the shared secret key);
CBC-MAC [60], SHA1-HMAC-80 [8] and UMAC [11] are examples of MACs that are believed to be secure.

8.2.3 Digital Signatures

$\mathsf{SIG}_{User}()$ will denote the signing function and VER_User() the verification function where User specifies the party that signs;
In order to simplify the pseudo-code, we will abstract out many of the details (e.g. key storage/retrieval, certificate checking, etc.).

Many digital signature schemes exist in the literature (see chapter 11 of [47] for examples) note however that DSS – the Data Signature Standard [62], Elgamal [30] and RSA [2, 43] are by far the most popular.

8.2.4 Public Key Certificates

Often used standards for certificates include PGP [19] and X.509 [36]. All dough PGP and X.509 both have the same IETF standard status, PGP is simple to use, whereas X.509 is constantly changing and very hard to comply with in practice. In order to simplify the pseudo-code, the certification and certificate verification mechanisms are implicitly (and properly) carried out.

8.2.5 Pseudo-Random Number Generation

PRNG(min,max);

Returns a pseudo-random number in the range [min,max]. There are two operation modes:

1. The random seed[21] is specified by the programmer.
2. The random seed is chosen from an entropy pool [22].

The pseudo-random numbers must be chosen extremely carefully because systems can be broken if inadequate pseudo-random functions or badly chosen seed are used (see for example [32]). We recommend the use of Yarrow [40], since its design is based on many years of research and experience [41] and because it's easy to use (the programmer doesn't need to provide a seed for example).

8.2.6 Prime Number Generators

We use safe primes, i.e. primes of the form $p = 2q + 1$ (where q is prime) such that $g = 2$ is a generator of a subgroup of order q. We also suggest that a proof that the primes have not been chosen maliciously be given (i.e. we want kosherized primes). This choice helps satisfy requirements 2, 3 and 4 of section 7.1.
[51] suggests the use of "special" safe primes which are used in the description of IKE [35] (a candidate DH protocol for IPsec). They have properties that enable efficient modular computations:

- The 64 high order bits are set to 1, so that the trial quotient digit in the classical remainder algorithm can always be set to 1.
- The 64 low order bits are also set to 1, which enables speed ups of Montgomery style remainder algorithms.
- The middle bits are taken from the binary expansion of π which provides a weak form of kosherization.
- $g = 2$ is a generator of a subgroup of order $(p - 1)/2$ (g has prime order).

These primes can be found in Appendix E.2 (1024 bits) and E.5 (1536 bits) of [51] and can be used in a DH scheme that needs only one publicly known prime.

If lots of primes need to be generated *and* efficiency is an important requirement, we suggest using Lim-Lee primes [45] which are used in many cryptographic libraries (e.g. PGP [52],GNU PG [31] and Gutmann's cryptlib [33]). These primes have the form $p = 2q_1q_2 \dots q_n + 1$ where the q_is are large (for all $i \in \{1, \dots, n\}$). The generator can be taken to generate some prime order subgroup (e.g. of order q_i, for some i). A drawback to this method is that the range of values exponents can take is limited (i.e. exponents are taken modulo q_i instead of q) which restricts the range of

[21] an initialization value for pseudo- random functions.
[22] a pool of "random" bits – these depend on unpredictable events (e.g. mouse movements).

possible DH secret keys. Also note that the probability that a small generator generates a "good" subgroup is lower than for safe primes.

We now give pseudo-code for generating kosherized safe primes of the form $p = 2q + 1$ such that $g = 2$ generates a subgroup of order q.

Pseudo-Code:

Generating Kosherized safe Primes

```
/*
 *   GOAL: generate a 1024 bit Kosherized safe prime
 *
 *   GIVEN:
 *   -table containing the first NUMPRIMES primes -> prime_table[]
 *   -a probabilistic primality test -> Miller-Rabin(iter, n)
 *                                      iter: number of iterations
 *                                      n: the integer to test for primality.
 *                                      Returns TRUE if success, else FALSE.
 *   Constants:
 *     NUMB_ITER = 3
 *     NUMPRIMES = 2056 or a more optimal number (see Comments bellow code)
 */



/***** compute the random bits *****/

  /* initialize the counter */
  ctr := 0;


restart:

  /*
   * instead of choosing a new random seed each time, we use the same one
   * by trying it out 10 different ways.
   */
  ctr = ctr mod 30;
  if (ctr == 0) {
    SEED = PRNG(0, 2^512);  // 2^512 is 2 to the power 512
  }
  U = HS(SEED) || HS((SEED + ctr) mod 2^512) || HS((SEED + ctr + 1) mod 2^512)
              || HS((SEED + ctr + 2) mod 2^512) || HS((SEED + ctr + 3) mod 2^512)
              || HS((SEED + ctr + 4) mod 2^512);


/*
 *  We want to derive a 1023 bit odd value for q from U.
 */


  q := U<1,1023>;     //set q to be the first 1023 least significant bits of U
  q<1,1> := 1;        //set the least significant bit of q to 1 (odd)
  q<1023,1023> := 1;  //set the most significant bit of q to 1 (size)
```

```
/*
 *  check to see if q is not obviously composite. If it is composite, we would
 *  restart and pick another U.
 */

for (i = 0; i < NUMPRIMES; i ++) {
  if (div(q, prime_table[i])) == 0) {
     goto restart;  //failure, start over
  }
}


p = 2*q + 1;   // these are regular multiplication and addition (not modular)

/*
 *  check to see if 2*q + 1 is not obviously composite.  If it is composite,
 *  we would restart and pick another U.
 */

for (i = 0; i < NUMPRIMES; i ++) {
  if (div(p, prime_table[i])) == 0) {
     goto restart;  //failure, start over
  }
}

/*
 *  Verify that g = 2 is a generator of the subgroup of order
 *  q (see Comments below for the reasoning).  If it's not, than
 *  restart and pick another U.
 */

if (p != 7 mod 8) {
  goto restart;
}

/*
 *  We iterate, alternatively calling Miller-Rabin on p and q.
 *  This improves the efficiency on average.
 *
 */

for (i = 0; i < NUMB_ITER; i++) {
  if (Miller-Rabin(1, q) == 0) {
    goto restart;
  }
  if (Miller-Rabin(1, p) == 0) {
    goto restart;
  }
}

return p, q;
```

Comments:

- The algorithm takes on input a table of the first NUMPRIMES prime integers. The value of NUMPRIMES can be computed for optimal efficiency: Let t_E denote the time for a full modular exponentiation, and let t_D be the time for ruling out one small prime as divisor, then the value of NUMPRIMES that minimizes the expected running time of the preceding pseudo-code is NUMPRIMES $= t_E/t_D$. If for any reason this number can't be practically computed or if performance is not crucial, we recommend using NUMPRIMES $= 2056$ (if your hardware has at least 8 bit registers) such as used in open-SSL [50] for example.

- The pseudo-code uses an algorithm called Miller-Rabin, we do not discuss this protocol, one could use the one provided in a trusted cryptographic library ([50], [31], [33] for example) or see for example [47] for pseudo-code.

- For the resulting algorithm to be defined as a *robust* primalty test[23] and considering efficiency, one should choose NUMB_ITER to be 3 (see [24] for exact details).

- On kosherization: note that interested readers can verify that the primes haven't been chosen maliciously by simply taking the SEED, executing the previous protocol and verifying that the prime obtained equals p.

- We used the fact that $p = 7 \bmod 8 \Rightarrow g = 2$ generates an order q subgroup.
  This can be prooven by the following facts:
  $p = 7 \bmod 8 \iff^{24}$ 2 is a quadratic residue[25] mod $p \implies 2^{\frac{1}{2}}$ exists $\implies 2^{((p-1)/2)} = (2^{\frac{1}{2}})^{p-1} = ^{26}1 \pmod p$
  $\iff^{27}$ $g = 2$ generates an order q subgroup.

Notice that this choice of parameters p and g makes many of the attacks mentioned previously ineffective.

8.3 Data Structures

Several data structures are now presented (those that aren't presented here don't require any special explanations).

8.3.1 DH parameters (DH_PARAM)

```
structure DH_PARAM {
  BIGNUM p;                  //1024 bit prime
  BIGNUM q;                  //prime q such that p = 2q + 1;
  BIGNUM g;                  //order q subgroup generator
  RANGE range_max;           //maximum secret key (e.g. x) value
  RANGE range_min;           //minimum secret key value
};
```

The DH_PARAM data structure stores the DH system parameters.

We recommend using range_min equal 2^4 and range_max equal 160 bits (DSS [62] uses a 1024-bits p and 160-bits q).

8.3.2 Party Identifier (ID)

structure ID;

The ID structure stores identifying information about parties. For example, it could contain :

[23] that is, a test that has success greater than $\frac{1}{2^{80}}$.

[24] see for example Fact 2.146 in [47]. We can eliminate the case where $p = 1 \pmod 8$ since p and q are prime. (left as an exercice.)

[25] $x \in \mathbb{Z}_p^*$ is quadratic residue of $\mathbb{Z}_p^*$ iff there exists a $y \in \mathbb{Z}_p^*$ such that $y^2 = x \bmod p$. If no such y exists, x is called a quadratic non-residue.

[26] the equality comes from the fact that $p - 1$ is the order of the group.

[27] 2's order is either 2, q or $p - 1$ (Lagrange's theorem, section 2.1.3). If $2^q = 1 \pmod p$, then 2's order can only be 2 or q, but the only elements of order 2 are 1 and $p - 1$.

- (Unique) Identifiers.
- Certificates (optional).
- Types of protocols (and parameters) he can participate in.
- Location (e.g. IP address and port number).

8.3.3 Session Key Parameters (SESSION_KEY_PARAM)

structure SESSION_KEY_PARAM;
This structure is used to specify how the shared DH key will be used. For example, the structure might store information such as:

- Key derivation function specification.
- Symmetric encryption or MAC used with the session(s) key(s).
- Version numbers.

8.3.4 Message Identifiers (TAG)

These are used to identify the message types (we suspect these will be defined using the #define construct).

8.4 Send and Receive Primitives

8.4.1 Send Message

send_{source,destination}(...);

sends a message consisting of all the arguments from "source" to "destination".

8.4.2 Receive Message

receive_{source,destination}(...);

The arguments sent by "source" will be received by "destination".

N.B. "source" and "destination" give just enough information so that a message can be sent. Typically, this information will consist of an IP address and of a port number both of which are specified in IP packets (for recipient *and* sender). Notice that if a participant's connection is made through a mix-network, this information doesn't compromise privacy. "source" and "destination" should be *unique* – exactly one party (process, thread, etc) is associated with a "destination" (or "source").

8.5 Authenticated Ephemeral DH Key Agreement Protocol – providing forward secrecy

Here the two participants, Alice and Bob, want to frequently set up new session keys. Without loss of generality, we assume that Alice initiates the protocol. We assume that the tags INIT, ACCEPT and CONFIRM have been defined (e.g. #define INIT 1). Variables are defined in order to avoid any confusion.

8.5.1 Comments

The parameters could be specified only in the last message however this means that Bob will have to wait before computing the session key. In the pseudo-code presented above, Bob can start computing the session key as soon as he has computed the shared DH secret key (he doesn't have to wait for the CONFIRM message).

The two signatures sign messages that contain information about the signer and so are "self-signed". This means that the CA doesn't necessarily need to check that the applicant actually knows the secret key associated with the public key.

If there is a substantial gain in efficiency, only the most significant bits of gy} need to be hashed in the last message (we should take enough bits so that the chance of using them again is extremely small).

The last message (CONFIRM) is needed in order to prevent replay attacks.

If denial of service attacks need to be prevented, we propose using a SYN cookie[28] type mechanism (need to verify that the party attempting to connect is allowed to) when establishing the connection. This solution isn't presented in detail since the code depends low level networking protocols (note that SYN cookies are now a standard part of Linux). For half-certified DH, where the non-certified side may (or must) be anonymous, one would need some stronger type of denial of service preventions such as Client Puzzles ([38]).

9 Conclusion

This work has attempted to present cryptographic protocol designers with the most important security issues related to the DH protocol. In doing so, we have addressed the shortcomings of the other approaches to secure cryptographic protocol design. It is hoped that documents with a form similar to this one but for different cryptographic protocols will be produced. This would be a large step towards assuring cryptographic protocol security in real-world settings (i.e. not just theoretical settings).

References

[1] ABADI, M., AND GORDON, A. D. A calculus for cryptographic protocols: The spi calculus. *Information and Computation 148*, 1 (10 Jan. 1999), 1–70.

[2] ADLEMAN, L., RIVEST, R. L., AND SHAMIR, A. A method for obtaining digital signature and public-key cryptosystems. *Communication of the ACM 21*, 2 (1978).

[3] AGENCY, N. S. Skipjack and kea algorithm specification (version 2.0), May 1998. Also available at *http://csrc.nist.gov/encryption*.

[4] ANDERSON, R., AND NEEDHAM, R. Programming satan's computer. In *Computer Science Today: Recent Trends and Developments* (1995), J. van Leeuwen, Ed., vol. 1000 of *Lecture Notes in Computer Science*, Springer.

[5] ANDERSON, R., AND NEEDHAM, R. Robustness principles for public key protocols. In *Advances in Cryptology – CRYPTO ' 95* (1995), D. Coppersmith, Ed., Lecture Notes in Computer Science, Springer-Verlag, Berlin Germany.

[6] ANDERSON, R., AND VAUDENAY, S. Minding your p's and q's. In *Advances in Cryptology—ASIACRYPT '96* (Kyongju, Korea, 3–7 Nov. 1996), K. Kim and T. Matsumoto, Eds., vol. 1163 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 26–35.

[7] BACK, A. Hashcash. http://www.cypherspace.org/ adam/hashcash/, mar 1997.

[8] BELLARE, M., CANETTI, R., AND KRAWCZYK, H. HMAC: Keyed-hashing for message authentication, Feb. 1997.

[9] BELLARE, M., CANETTI, R., AND KRAWCZYK, H. Modular approach to the design and analysis of key exchange protocols. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC-98)* (New York, May 23–26 1998), ACM Press, pp. 419–428.

[10] BELLARE, M., AND ROGAWAY, P. Entity authentication and key distribution. In *Advances in Cryptology – CRYPTO ' 93* (1994), D. R. Stinson, Ed., vol. 773 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin Germany.

[28] see http://cr.yp.to/syncookies.html.

[11] BLACK, J., HALEVI, S., KRAWCZYK, H., AND KROVETZ, T. UMAC: Fast and secure message authentication. In *Advances in cryptology — CRYPTO '99: 19th annual international cryptology conference, Santa Barbara, California, USA, August 15–19, 1999 proceedings* (Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1999), M. Wiener, Ed., vol. 1666 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 216–233.

[12] BLAKE-WILSON, S., AND MENEZES, A. Authenticated Diffie-Hellman key agreement protocols. In *Fifth Annual Workshop on Selected Areas in Cryptography (SAC '98)* (1999), Lecture Notes in Computer Science, Springer Verlag, pp. 339–361.

[13] BLEICHENBACHER, D. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. *Lecture Notes in Computer Science 1462* (1998), 1–??

[14] BLEICHENBACHER, D., KALISKI, B., AND STADDON, J. Recent results on PKCS #1 RSA encryption standard. *RSA Laboratories' Bulletin*, 7 (1998).

[15] BONEH, D., AND VENKATESAN, R. Hardness of computing the most significant bits of secret keys in Diffie-Hellman and related schemes (extended abstract). In *Advances in Cryptology—CRYPTO '96* (18–22 Aug. 1996), N. Koblitz, Ed., vol. 1109 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 129–142.

[16] BRADEN, R. RFC 1644: T/TCP — TCP extensions for transactions functional specification, July 1994. Status: EXPERIMENTAL.

[17] BURMESTER, M. On the risk of opening distributed keys. In *Advances in Cryptology – CRYPTO ' 94* (1994), Y. G. Desmedt, Ed., vol. 839 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin Germany, pp. 308–317.

[18] BURROWS, M., ABADI, M., AND NEEDHAM, R. A logic of authentication. *ACM Transactions on Computer Systems 8*, 1 (Feb. 1990), 18–36.

[19] CALLAS, J., DONNERHACKE, L., FINNEY, H., AND THAYER, R. RFC 2440: OpenPGP message format, Nov. 1998. Status: PROPOSED STANDARD.

[20] CERT. Advisory ca-96.21: Tcp syn flooding and ip spoofing attacks, 24 September 1996.

[21] CHAUM, D. Blind signatures for untraceable payments. In *Advances in Cryptology: Proceedings of Crypto 82* (23–25 Aug. 1982), D. Chaum, R. L. Rivest, and A. T. Sherman, Eds., Plenum Press, New York and London, 1983, pp. 199–203.

[22] CHOR, B.-Z. *Two issues in public key cryptography: RSA bit security and a new knapsack type system.* ACM distinguished dissertations. MIT Press, Cambridge, MA, USA, 1986. Originally presented as the author's thesis (doctoral — MIT, 1985).

[23] DAEMON9. Project neptune. Phrack Magazine, 48(7): File 13 of 18, 8 November 1996. Available at www.fc.net/phrack/files/p48/p48-13.html.

[24] DAMGÅRD, I., LANDROCK, P., AND POMERANCE, C. Average case error estimates for the strong probable prime test. *Mathematics of Computation 61*, 203 (July 1993), 177–194.

[25] DEN BOER, B., AND BOSSELAERS, A. Collisions for the compression function of MD5. In *Advances in Cryptology—EUROCRYPT 93* (23–27 May 1993), T. Helleseth, Ed., vol. 765 of *Lecture Notes in Computer Science*, Springer-Verlag, 1994, pp. 293–304.

[26] DIERKS, . T., AND ALLEN, C. RFC 2246: The TLS protocol version 1, Jan. 1999. Status: PROPOSED STANDARD.

[27] DIFFIE, W., AND HELLMAN, M. New directions in cryptography. *IEEE Transactions on Information Theory 22* (1976), 644–654.

[28] DOBBERTIN, H., BOSSELAERS, A., AND PRENEEL, B. RIPEMD-160: A strengthened version of RIPEMD. In *Fast Software Encryption — Cambridge Workshop* (Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1996), D. Gollman, Ed., vol. 1039 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 71–82. (An updated and corrected version is available at ftp.esat.kuleuven.ac.be, directory /pub/COSIC/bosselae/ripemd/.).

[29] DWORK, C., AND NAOR, M. Pricing via processing or combatting junk mail. *Lecture Notes in Computer Science 740* (1993), 139–147.

[30] ELGAMAL, T. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in Cryptology: Proceedings of CRYPTO 84* (19–22 Aug. 1984), G. R. Blakley and D. Chaum, Eds., vol. 196 of *Lecture Notes in Computer Science*, Springer-Verlag, 1985, pp. 10–18.

[31] FOUNDATION, F. S. Gnu privacy guard. http://www.gnupg.org/.

[32] GOLDBERG, I., AND WAGNER, D. Randomness and the Netscape browser. *Dr. Dobb's Journal of Software Tools 21*, 1 (Jan. 1996), 66, 68–70.

[33] GUTMANN, P. http://www.cs.auckland.ac.nz/ pgut001/cryptlib/.

[34] GUTMANN, P. Secure deletion of data from magnetic and solid-state memory. In *6th USENIX Security Symposium* (San Jose, California, July 1996), USENIX.

[35] HARKINS, D., AND CARREL, D. RFC 2409: The Internet Key Exchange (IKE), Nov. 1998. Status: PROPOSED STANDARD.

[36] HOUSLEY, R., FORD, W., POLK, W., AND SOLO, D. RFC 2459: Internet X.509 public key infrastructure certificate and CRL profile, Jan. 1999. Status: PROPOSED STANDARD.

[37] HUNGERFORD, T. W. *Algebra*. Holt, Rinehart and Winston, New York, 1974.

[38] JUELS, A., AND BRAINARD, J. Client puzzles: A cryptographic defense against connection depletion attacks. In *NDSS '99 (Networks and Distributed Security Systems)* (2000), S. Kent, Ed., pp. 151–165.

[39] KARN, P., AND SIMPSON, W. A. The photuris session key management protocol. Internet Draft, Dec. 1995. Version 0.8, expires June 96.

[40] KELSEY, J., SCHNEIER, B., AND FERGUSON, N. Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator. In *Sixth Annual Workshop on Selected Areas in Cryptography* (Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1999), Springer-Verlag, p. ????

[41] KELSEY, J., SCHNEIER, B., WAGNER, D., AND HALL, C. Cryptanalytic attacks on pseudorandom number generators. *Lecture Notes in Computer Science 1372* (1998), 168–188.

[42] KOCHER, P. Cryptanalysis of Diffie-Hellman, RSA, DSS, and other cryptosystems using timing attacks. In *Advances in cryptology, CRYPTO '95: 15th Annual International Cryptology Conference, Santa Barbara, California, USA, August 27–31, 1995: proceedings* (Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1995), D. Coppersmith, Ed., vol. 963 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 171–183.

[43] LABORATORIEW, R. *PKCS #1 - The Public-Key Cryptography Standard*. RSA Data Security Inc., September 1998.

[44] LENSTRA, A. K., AND VERHEUL, D. E. R. Selecting cryptographic key sizes. urlhttp://www.cryptosavvy.com/, Nov. 1999. Shorter version of the report appeared in the proceedings of the Public Key Cryptography Conference (PKC2000) and in the Autumn '99 PricewaterhouseCoopers CCE newsletter.

[45] LIM, C. H., AND LEE, P. J. A key recovery attack on discrete log-based schemes using a prime order subgroup. In *Advances in Cryptology – CRYPTO ' 97* (Aug. 1997), B. S. K. Jr., Ed., Lecture Notes in Computer Science, Springer-Verlag, Berlin Germany, pp. 249–263.

[46] MAUGHAN, D., SCHERTLER, M., SCHNEIDER, M., AND TURNER, J. RFC 2408: Internet Security Association and Key Management Protocol (ISAKMP), Nov. 1998. Status: PROPOSED STANDARD.

[47] MENEZES, A. J. A. J., VAN OORSCHOT, P. C., AND VANSTONE, S. A. *Handbook of applied cryptography*. The CRC Press series on discrete mathematics and its applications. CRC Press, 2000 N.W. Corporate Blvd., Boca Raton, FL 33431-9868, USA, 1997.

[48] MITCHELL, C. Limitations of challenge-response entity authentication. *Electronics letters*, 25 (August 1989), 1195–1196.

[49] ODLYZKO, A. M. Discrete logarithms: The past and the future. *Designs, Codes, and Cryptography 17* (1999).

[50] OPEN-SSL. http://www.openssl.org/.

[51] ORMAN, H. RFC 2412: The OAKLEY Key Determination Protocol, Nov. 1998. Status: INFORMATIONAL.

[52] PGP, I. http://www.pgpi.org/.

[53] POHLIG, S., AND HELLMAN, M. An improved algorithm for computing discrete logarithms over GF(p) and its cryptographic significance. *IEEE Transactions on Information Theory 24* (1978), 106–110.

[54] POLLARD, J. M. Monte Carlo methods for index computation $(\mathrm{mod} p)$. *Mathematics of Computation 32*, 143 (July 1978), 918–924.

[55] RIVEST, R. RFC 1321: The MD5 message-digest algorithm, Apr. 1992. Status: INFORMATIONAL.

[56] RIVEST, R. L., AND SHAMIR, A. How to expose an eavesdropper. *Communications of the Association for Computing Machinery 27*, 4 (Apr. 1984), 393–395.

[57] SCHNORR, C. P. Efficient identification and signatures for smart cards. In *Advances in Cryptology—EUROCRYPT 89* (10–13 Apr. 1989), J.-J. Quisquater and J. Vandewalle, Eds., vol. 434 of *Lecture Notes in Computer Science*, Springer-Verlag, 1990, pp. 688–689.

[58] SHOSTACK, A. http://www.homeport.org/ adam/crypto/.

[59] SILVERMAN, R. A cost-based security analysis of symmetric and asymetric key lengths. RSA Laboratories Bulletin, april 2000.

[60] STANDARDS, F. I. P. Des modes of operation. Tech. Rep. FIPS PUB 81, National Institute of Standards and Technology, december 1980.

[61] STANDARDS, F. I. P. Secure hash standard. Tech. Rep. FIPS PUB 180-1, National Bureau of Standards, 1995.

[62] STANDARDS, F. I. P. Digital signature standard (dss). Tech. Rep. FIPS PUB 186, National Bureau of Standards, 1998.

[63] STINSON, D. R. *Cryptography: Theory and Practice*. CRC Press, 1995.

[64] VAN OORSCHOT, P. C., AND WIENER, M. J. On Diffie-Hellman key agreement with short exponents. In *Advances in Cryptology – EUROCRYPT ' 96* (1996), U. Maurer, Ed., Lecture Notes in Computer Science, Springer-Verlag, Berlin Germany, pp. 332–343.

[65] WAGNER, D., AND SCHNEIER, B. Analysis of the SSL 3.0 protocol. Technical report, Counterpane Systems, 101 East Minnehaha Parkway, Minneapolis, MN 55419, 1996. Also published in *The Second USENIX Workshop on Electronic Commerce Proceedings*, USENIX Press, November 1996, pp. 29–40.

A Standards

A.1 PKCS #3

RSA security [29] has published a suite of cryptography standards which are called Public Key Cryptography Standard (PKCS). PKCS #3 deals with the DH protocol. Unfortunately, it does not help the protocol designer construct a secure version because it only specifies data formats.

A.2 ANSI X9.42 – Agreement of Symmetric Algorithm Keys Using Diffie-Hellman

working draft may 1998

A.3 IETF RFC 2522 – Photuris: Session-Key Management Protocol

march 1999

A.4 ANSI X9.63 – Elliptic Curve Key Agreement and Key Transport Protocols

working draft July 1998

A.5 IEEE P1363 – Standard Specifications for Public-Key Cryptography

working draft July 1998

A.6 ISO/IEC 11770-3 – Information Technology - Security Techniques - Key Management - Part 3: Mechanisms Using Asymmetric Techniques

draft (DIS), 1996

A.7 SKIPJACK and KEA algorithm specification

from FORTEZZA may 1998.

A.8 The Internet Key Agreement (IKE)

RFC 2409 November 1998.

A.9 The OAKLEY key Determination Protocol

RFC 2412 November 1998.

A.10 The TLS Protocol: Version 1.0

RFC 2246 January 1999

[29] see http://www.rsasecurity.com