Chapter 12:
YYGen: A Backdoor-Resistant RSA Key Generator*

Adam L. Young and Moti M. Yung

Abstract

In this chapter we present an algorithm that generates RSA key pairs in such a way that the subliminal channel in the RSA modulus n is significantly reduced. We refer to this key generator as YYGen. YYGen takes as input an RSA public exponent e that we assume is fixed and shared among all users. The key generation and verification process is as follows. A user generates a key pair, sends the public key plus auxiliary information to a verifier, and the verifier is able to verify using algorithm YYVer that the subliminal channel is reduced in n. Our solution is reminiscent of the FIPS 186-2 parameter generation method that helps guard against backdoors in DSA parameters. A disadvantage of the solution is that it does not guarantee subliminal-freeness. However, an advantage is that the solution is extremely lightweight and can be easily deployed in hardware (smartcards) and software. Our solution heuristically foils all published asymmetric backdoors in RSA key generation that we have found that utilize fixed e. For this reason we believe that YYGen and YYVer form a desirable alternative to typical RSA key generators in sensitive infosec environments.

1 Introduction

Every implementation of RSA key generation that we have seen to date exhibits a gaping huge subliminal channel in the public keys that are generated. This is not a problem when the the manufacturer of the key generation device is trustworthy, or when the device in question is verified for correctness.[1]

*If this file was obtained from a publicly accessible website other than the website www.cryptovirology.com then (1) the entity or entities that made it available are in violation of our copyright and (2) the contents of this file should therefore not be trusted. Please obtain the latest version directly from the official Cryptovirology Labs website at: http://www.cryptovirology.com.

[1]This type of verification can be difficult if not impossible in tamper-resistant chips and the chip may be destroyed during verification.

However, when these two situations are not the case, there is an attractive opportunity for a malicious designer to insert an asymmetric backdoor into the key generator.

In Crypto '96 [16] (see also [17, 18]) we introduced the notion of an asymmetric backdoor and showed how to build one into the RSA key generation algorithm. The backdoor has several properties. The primary novelty is that it was the first *asymmetric* backdoor: even if the backdoor algorithm becomes public, only the designer can use the backdoor, no one else can. It also has the property that when implemented in a black-box device, the distribution over key pairs that contain the backdoor is computationally indistinguishable from the distribution over "normal" RSA key pairs (as induced by the backdoor key generator and the normal key generator, respectively). The construction has the property that the backdoor information is encoded into the upper half of the bits of the RSA modulus that is generated.

There exists a heuristic method for foiling the asymmetric backdoor [14]. Informally, the idea is to "plug-up" the subliminal channel with the cryptographic hash of a randomly selected preimage and give the preimage to a verifier. The "plug-up" operation (omitting details) amounts to essentially making the upper half of the bits of the RSA modulus n be the hash. Another interpretation of this is that the user that generates a key pair can only indirectly select the upper order bits of n, and do so in a very restricted fashion. In some sense, it is the hash function that directly "chooses" the upper order bits of n.

Our hope in publishing [14] was to draw more attention to the defensive measures that exist and also lay some of the formal groundwork for showing that our backdoor-resistant RSA key generator is sound. We had hoped that standardization bodies might take action as a result, but this has not happened so far.

There exists a NIST approved method for generating DSA parameters that relies on the SHA1 function. This method is described in FIPS 186-2 (see Appendix 2 of [8]). The method allows an entity to generate group parameters in such a way that a verifier can check that SHA1 was used according to the method. When used, this method heuristically implies that the group parameters could not have been "cooked" to contain a weakness or backdoor. However, at the time of our Crypto '96 paper there was no analogous method for RSA key generation. Our work in [16, 14] paves the way for such an analogue.

As illustrated in Chapter 10 of this book, Kleptography has matured from a theoretical science into an experimental science. We have placed our

experimental elliptic curve backdoor in RSA key generation on the Internet for cryptologists to analyze/break/study. This backdoor design is based on [15]. It therefore behooves us to do the same for our proposed backdoor resistant key generator.

In this chapter we move further down the path of defending against high-bandwidth asymmetric backdoors in RSA key generation by presenting an experimental implementation of a backdoor-resistant RSA key generator. Our results show that the generator is quite fast on modern computers.

We restrict our attention to RSA key generators that employ a fixed RSA exponent e that is shared among users. It is extremely common for $e = 2^{16} + 1$ to be used in practice. All of the backdoors in [2] except for the one in Section 5 fail to function in this case. Our backdoor-resistant RSA key generator defends against the backdoor in Section 5. Also, we remark that all of the backdoors in [2] are symmetric rather than asymmetric.

The solution we describe in this chapter is no silver bullet. It is trivial to construct a channel in `YYGen` that leaks $O(\log(\log(n)))$ subliminal bits in n. Also, the preimage must be handled carefully since it is a pure subliminal channel.

However, our construction is an improvement that we believe constitutes due diligence for organizations concerned about information security. We believe that an organization that uses our approach to defending against RSA backdoors will be taking a solid step towards defending itself from the insider threat and also from the outsourced provider threat.

2 Background

Gus Simmons pioneered the notion of subliminal channels in cryptosystems. His first hypothetical application was a way to Trojanize the Salt II treaty verification devices to secretly reveal which silos were armed with Minute-Man missiles and which ones were not [13]. This leakage occurred through an authentication scheme based on symmetric cryptographic primitives. He also formulated the Prisoners' Problem as a way to exemplify the threat of subliminal channels within the framework of a more traditional digital signature scheme [12].

Subliminal channels have been identified in a great many cryptographic algorithms and protocols. Yvo Desmedt observed that it is possible to choose one-half of the bits of the product of two primes explicitly when the algorithm chooses any two large prime numbers [3]. Further work was conducted by Lenstra who reviewed and generalized a method to generate RSA moduli

with a predetermined portion [6].

There has been formal work on trying to eliminate subliminal channels in protocols and algorithms. Desmedt, Goutier, and Bengio provided a solution to avoid a subliminal channel in the Fiat-Shamir protocol [5]. Research has also be conducted to cryptanalyze existing subliminal-free solutions [4].

There are many results on subliminal channels and defenses against subliminal channels that have not been mentioned here. We hope that these references will serve as a good starting point for reading about work in this area.

3 YYGen and YYVer

The approach to reducing the subliminal channel in [14] employs a random oracle. In practice such an idealized function must be instantiated with an existing cryptographic primitive. This leaves the practical problem of selecting a secure primitive. When considering the specific problem of generating 1024-bit RSA keys with resistance to subliminal channels, we believe that the SHA-512 function [9] provides an attractive replacement for a random oracle. We outline some of the merits of this choice below.

1. SHA-512 is part of the FIPS standards and has undergone significant scrutiny.
2. SHA-512 is deployed in the OpenSSL cryptographic library which is available as open-source.
3. The range of the SHA-512 function is $\{0,1\}^{512}$. A SHA-512 hash value therefore fits perfectly in the upper half of a 1024-bit RSA modulus.
4. SHA-512 has 8 state variables that are each 64-bits in length. This provides a good "spread" of the preimage value across the image, which in our application is used to form the upper 512 bits of the RSA modulus.

The following notation and functions are used to describe `YYGen` and `YYVer`. The operator $>>$ is used for right-shifting a bit string. For example, $a >> b$ denotes right-shifting the bit string a by b bits and returning the resulting string. So, $1011 >> 2$ is the bit string 10. We define `IsNotPrime`(p) to be a primality testing algorithm. It returns true if it determines that p is not prime, otherwise it returns false. It is required that this predicate have a very small probability of error. The function `gcd` returns the greatest

common divisor of two integers. So, $\texttt{gcd}(a, b)$ is the gcd of the integers a and b. The function $\texttt{SHA512}(s)$ returns the SHA-512 hash of the input bit string s.

It is typical for an RSA key generator to always produce moduli that are an integral multiple of 8 bits in length. For 1024-bit moduli, this implies that the bit in bit position 1023 is always 1. The bit positions range from 0 to 1023, inclusive. Given this practice we impose the requirement that our generator produce 1024-bit RSA moduli n. Our RSA key generator uses the hash function `Hash511` defined below.

$\texttt{Hash511}(x)$:
1. compute $y = \texttt{SHA512}(x)$
2. overwrite the most significant bit of y with 1
3. output y

The purpose of setting the most significant bit to 1 in the hash is as follows. This hash will be embedded in the upper half of the bits of n. Placing a 1 here will ensure that n is a 1024-bit modulus.

We decided not to design an RSA key generator that incrementally searches for primes. The speed gain that results from incremental search is not very significant given how powerful general purpose computers are these days and when considering that RSA key generation is a one-time operation. However, in portable devices it may be the case that incremental search is desirable. We may provide an alternate design at a later time if this use-case seems important.

So, `YYGen` below does not incrementally search for the primes p and q. The value n_c stands for "candidate n". This value is generated after p is found but before q is determined.

$\texttt{YYGen}(p, q, s, e)$:
Input: RSA exponent e
Output: RSA primes p and q, $s \in \{0,1\}^{1024}$
1. choose $r \in_R \{0,1\}^{509}$ and set $p = 11 \,||\, r \,||1$
2. if ($\texttt{IsNotPrime}(p) =$ true) then goto step 1
3. if $\texttt{gcd}(p-1, e) \neq 1$ then goto step 1
4. set $m = \lfloor \frac{2^{1024+512-1}}{p} \rfloor$
5. choose $s \in_R \{0,1\}^{1024}$ and $r_c \in_R \{0,1\}^{512}$
6. let n_c be the integer corresponding to the binary string $\texttt{Hash511}(s) \,||\, r_c$
7. compute $t = m * n_c$
8. set $q = t >> (1024 + 512 - 1)$

9. if $(q >> 510) \neq 11$ then goto step 5
10. if (IsNotPrime(q) = true) then goto step 5
11. if gcd$(q-1, e) \neq 1$ then goto step 5
12. output (p, q, s)

A naive approach is to iteratively compute n_c/p as we run through the candidates in search of q. This is naive since it performs long divisions inside the loop when divisions are not needed inside the loop. A faster approach is to precompute a division by p and then do multiplication followed by right shifting in the loop. For more information on the relationship between computing divisions and multiplications see Chapter 8 of Aho-Hopcroft-Ullman [1].

YYGen solves the following equation for q and r for each candidate n_c.

$$n_c = pq + r \tag{1}$$

When a valid RSA prime q is found, the following equation shows the relationship between p and q that are output and n_c.

$$n = n_c - r = pq \tag{2}$$

The subtraction of the remainder r in equation 2 may cause a borrow bit to be taken from the upper 512 bits of n_c. This means that the Hash511 hash might not appear verbatim in the upper 512 bits of n even though it appears verbatim in n_c. For this reason the verification algorithm YYVer must add 1 to handle the case of a borrow bit being taken.

YYVer(n, s):
Input: 1024-bit RSA modulus n, $s \in \{0,1\}^{1024}$
Output: $r \in \{0, 1\}$
1. set $w =$ Hash511(s) and set $r = 0$
2. set $u_1 = n >> 512$
3. if $(u_1 = w)$ then set $r = 1$
4. else
5. set $u_2 = u_1 + 1$
6. if $(u_2 = w)$ then set $r = 1$
7. output r

Ignoring details, YYVer verifies that the hash of s is properly encoded in the upper order bits of n. It is imperative that programmers understand

how not to use s in practice. **The value s is a pure unadulterated subliminal channel**. It can only be disclosed to trusted verifiers and should under no circumstances be made generally accessible. The purpose of s is to give a trusted verifier evidence (albeit incomplete) that `YYGen` was used to compute p and q given e, and to show that well-publicized kleptographic attacks could not have occurred during key generation.

4 Creating the Program

The program `yygen` consists of the ANSI C source file `yygen.c`. We constructed a simple makefile to create `yygen`. The program was compiled using `gcc`. We used the Minimalist GNU for Windows development suite (MinGW) and employed the OpenSSL crypto library for its underlying crypto routines. In particular we used OpenSSL version 0.9.8d. Our program also utilizes the Microsoft Cryptographic API (MS CAPI) to seed the OpenSSL random number generator. Compiling the program results in the MS DOS command-line program `yygen.exe`.

5 Running the Program

One of the first things that `yygen` does is seed the OpenSSL random number generator. It obtains 512 random bytes using the Microsoft Cryptographic API call `CryptGenRandom` and then passes them to the OpenSSL function `RAND_seed`. The program then calls `RAND_status`. This OpenSSL function returns 1 if OpenSSL believes that it has been seeded with a sufficient number of bytes and it returns 0 otherwise.

```
"yygen" Copyright (c) 2005-2007 by
Moti Yung and Adam L. Young. All rights reserved.
Obtain latest version from www.cryptovirology.com.
This program uses API calls from the OpenSSL library.
This program is based on the following:
(1) A.  Young, "Mitigating insider threats to RSA
  key generation," CryptoBytes, RSA Laboratories,
  vol. 7, no. 1, Spring 2004.
(2) A. Young, M. Yung, "Malicious Cryptography: Exposing
  Cryptovirology", John Wiley & Sons, Feb., 2004.
(3) A. Young, M. Yung, "The Dark Side of Black-Box
  Cryptography, or: Should we trust Capstone?" Advances in
  Cryptology---Crypto '96, N. Koblitz (Ed.), LNCS 1109,
```

```
  pp. 89-103, 1996.
RAND_status() returned 1.
Commands:
Type (a) to generate an RSA key pair using YYGen.
Type (b) to use YYVer to verify a public key produced
         using the YYGen algorithm.
Type (c) to test RSA encryption and decryption using
         the RSA key pair.
Type (d) to do 10000 tests of YYGen followed by YYVer.

Enter command (a-d) :
```

This chapter will only cover commands (a) through (c). Command (d) is used for software development and testing purposes.

5.1 Generating an RSA Key Pair Using YYGen

Command (a) generates an RSA key pair using YYGen. The two RSA primes are written to rsaprivkey.txt. The RSA modulus n, exponent e, and the YYGen preimage s are written to rsapubkey.txt. All of these values are in the form of ASCII strings expressed as hexadecimal integers (leading zeros if present are dropped).

```
Enter command (a-d) : a
Generating and storing RSA key pair using YYGen.
All values are expressed in hexadecimal.
The SHA-512 preimage is the 128 byte array s[].
Generating RSA prime p...
Generating RSA prime q...
s[] = 00D9E33FBCEC167B945C1BE6C20387D1936BEF19A05A8757BA2A86854A
8B0F037E2095EA7D7AFE64ED10A71723A3697839D6BA3BD44F0BF7BBE7A03B65
2145347BC7B57245599FE7F403B7D3374F1AEBC14ACC913C8141A5E078F47196
16031B621C192A4DD38A26510B2638603B900CB5A5076FC3B517A2096AC1B10C
31F1C0
rsa->p = EB2B85EBEBFC810931059DE46AFEB07F8446A63EA204D3EE2A2CBFB
50D3AA753D1FEB78DFF77FBA230A320FFF71A360E30C3853AB57A9DD77763246
8C9398341
rsa->q = E7A4E36ED7AE84C6CAF93CCC54C65BF034035C103635CB7150062F3
89E25D95BEB8BAEEDA9C311D3D40E17EC7DD1AD1E6DFE65DEF29916CCAA58C34
44B16AB8D
rsa->e = 10001
rsa->n = D4CBBEA717A9A2D97AC20E7DC52B9D0B0607BA60AA0ED1882448DB9
```

```
1603452C7739999D6BCEA0F5686AC4451F4E30FC243C8DA990806A713C26C0B9
C3F024CB9AA8B408F51842D1654FB9A43E05DA40AEF3B06727AFE2E8C2348534
20BFA332B45BD113B7B8E5109A13041A4FA8F5F799F2C286110223272597B90C
991EFB5CD
Checking the correctness of the RSA key pair...
OpenSSL function RSA_check_key() returned 1. So, the
key pair should be properly formed.
Wrote preimage and RSA public key to "rsapubkey.txt".
Wrote RSA private key to "rsaprivkey.txt".
-----Memory Leaks displayed below--(shouldn't be any)---
-----Memory Leaks displayed above----------------------
```

The RSA primes and exponent are passed to a modified version of the OpenSSL function `rsa_builtin_keygen`. We changed this OpenSSL function to accept p and q as input rather than generate them randomly. The output is an RSA key pair data structure. Elements of this data structure enable OpenSSL to perform fast RSA decryption [10, 11]. As a precaution, we pass this data structure to the OpenSSL function `RSA_Check_Key` to verify its integrity. A result of 1 indicates a properly formed key pair structure.

5.2 Verifying the Public Key Using YYVer

Command (b) is used to verify that the public key in the file `rsapubkey.txt` has a reduced subliminal channel. The modulus n, public exponent e, and SHA-512 preimage s are read in from this file. The values n and s are passed to `YYVer` that returns 0 or 1. A return value of 1 indicates that the public key corresponds to the format defined by `YYGen`.

```
Enter command (a-d) : b
Loading RSA public key from file "rsapubkey.txt".
n = D4CBBEA717A9A2D97AC20E7DC52B9D0B0607BA60AA0ED1882448DB916034
52C7739999D6BCEA0F5686AC4451F4E30FC243C8DA990806A713C26C0B9C3F02
4CB9AA8B408F51842D1654FB9A43E05DA40AEF3B06727AFE2E8C234853420BFA
332B45BD113B7B8E5109A13041A4FA8F5F799F2C286110223272597B90C991EF
B5CD
e = 10001
Loading preimage s from file "rsapubkey.txt".
s[] = 00D9E33FBCEC167B945C1BE6C20387D1936BEF19A05A8757BA2A86854A
8B0F037E2095EA7D7AFE64ED10A71723A3697839D6BA3BD44F0BF7BBE7A03B65
2145347BC7B57245599FE7F403B7D3374F1AEBC14ACC913C8141A5E078F47196
```

```
16031B621C192A4DD38A26510B2638603B900CB5A5076FC3B517A2096AC1B10C
31F1C0
1 plus the upper 512 bits of n is the hash.
LoadYYGenPubKeyIsValid() returned 1.
The heuristic verification algorithm YYVer
determined that the public key is valid.
-----Memory Leaks displayed below--(shouldn't be any)---
-----Memory Leaks displayed above------------------------
```

In the example above, the upper half of the bits of n is not the SHA-512 hash of s. Upon dividing n_c by p in `YYGen`, a remainder $r < p$ is produced that causes $n_c - r$ to take a borrow bit from the upper 512 bits of n_c. To handle this case, `YYVer` adds 1 to the upper 512 bits of n and compares the result to `Hash511`(s). It turns out that these two values are equal in this example. This is why the output "1 plus the upper 512 bits of n is the hash" is issued.

When a borrow bit is not taken from the upper 512 bits of n_c, the output that is displayed by `yygen.exe` is different. In this case the output reads, "The hash is the upper 512 bits of n exactly".

It is worth issuing commands (a) and (b) in succession several times. After only a few tries one would expect to witness both of these cases.

5.3 Test RSA Encryption and Decryption

Command (c) is used to perform a simple verification of the RSA key pair. It loads the RSA public key from the file `rsapubkey.txt` and the RSA private key from the file `rsaprivkey.txt`.

```
Enter command (a-d) : c
Loading RSA public key from file "rsapubkey.txt".
n = D4CBBEA717A9A2D97AC20E7DC52B9D0B0607BA60AA0ED1882448DB916034
52C7739999D6BCEA0F5686AC4451F4E30FC243C8DA990806A713C26C0B9C3F02
4CB9AA8B408F51842D1654FB9A43E05DA40AEF3B06727AFE2E8C234853420BFA
332B45BD113B7B8E5109A13041A4FA8F5F799F2C286110223272597B90C991EF
B5CD
e = 10001
Loading preimage s from file "rsapubkey.txt".
s[] = 0D9E33FBCEC167B945C1BE6C20387D1936BEF19A05A8757BA2A86854A
8B0F037E2095EA7D7AFE64ED10A71723A3697839D6BA3BD44F0BF7BBE7A03B65
2145347BC7B57245599FE7F403B7D3374F1AEBC14ACC913C8141A5E078F47196
16031B621C192A4DD38A26510B2638603B900CB5A5076FC3B517A2096AC1B10C
```

```
31F1C0
Loading RSA private key from file "rsaprivkey.txt".
p  = EB2B85EBEBFC810931059DE46AFEB07F8446A63EA204D3EE2A2CBFB50D3
AA753D1FEB78DFF77FBA230A320FFF71A360E30C3853AB57A9DD777632468C93
98341
q  = E7A4E36ED7AE84C6CAF93CCC54C65BF034035C103635CB7150062F389E2
5D95BEB8BAEEDA9C311D3D40E17EC7DD1AD1E6DFE65DEF29916CCAA58C3444B1
6AB8D
d = 5367944F7BB7D28B79510C4B017809B2A3676E06AB40E9A179CF50B81744
0ADA00164934E090C0F8420ACD306E527CFFC07FBE2652FE00887F20C8203296
1D29C59586FA234AE4D8EED209E889D3254101F4CF91841F3043E9F5E78026E8
B70B3223E1C71391ED1FB32C1B67EF81579BFAB8A9520E97F779576278CF20F1
E101
Choosing m_1 randomly from Z_n^*...
plaintext m_1 = 982DF9FD84214497691B6179DBEA1F37D9758F364912FBBA
C4F5360BC15B4EC3309BAB7354A58DAC4587BC09B4CE172E25E4247E015E7553
FEF31F940CD914E84034F5EB6D5B77289C5529D2D2A9D77B118CD52EA9A564D0
57EC0D58E90122537CFF136A34DD1F8989D59655CA283CB2A5707F27E4E1188E
A4B481CD8DF43778
ciphertext c = m_1^e mod n = BBE7140B9273656CB2BC3608B9F718E509F
501300DF63D531B580328BB93FFA3BC41352A4DD641EE032702AAA64D1E33BB9
731E9B43E29A1D139D1A130AE6F0EE0C4E9696B2D6A4F4F909DDFD84EC57287F
EC67677C996E67837125570C772D7FF77E0EDDFC49DD6DD6E615AA635C918999
373B2C9C4350318DB327970B9F08
Now computing m_2 = c^d mod n...
plaintext m_2 = 982DF9FD84214497691B6179DBEA1F37D9758F364912FBBA
C4F5360BC15B4EC3309BAB7354A58DAC4587BC09B4CE172E25E4247E015E7553
FEF31F940CD914E84034F5EB6D5B77289C5529D2D2A9D77B118CD52EA9A564D0
57EC0D58E90122537CFF136A34DD1F8989D59655CA283CB2A5707F27E4E1188E
A4B481CD8DF43778
RSA decryption succeeded since m_2 = m_1.
-----Memory Leaks displayed below--(shouldn't be any)---
-----Memory Leaks displayed above-----------------------
```

Command (c) generates a random plaintext m_1 and then RSA encrypts it to get the ciphertext c. The value c is decrypted using the RSA private key. The command indicates whether or not RSA decryption succeeds.

6 Design Alternatives

As a precaution, the set of acceptable RSA primes can be restricted to use *strong primes*. A number p is a strong prime if $r \mid p-1$, $s \mid p+1$, and

$t \mid r - 1$ where r, s, and t are large primes [7]. By making both p and q in $n = pq$ be strong primes, certain cryptanalytic attacks will be thwarted.

Another alternative is as follows. Since s is a pure subliminal channel, it should not be made widely accessible. This implies that `YYGen` is not publicly verifiable. This problem can be addressed as follows.

Let H_1 and H_2 be cryptographic hash functions that output 512-bit hashes. We make the upper half of the bits of n_c be $\mathrm{H}_1(\mathrm{H}_2(s))$. Both the user that generates the key pair and the verifier (e.g., a certification authority) are given s. The value $\mathrm{H}_2(s)$ can be published in the digital certificate along with (n, e).

The addition of another level of hashing removes, for the most part, the subliminal channel in $\mathrm{H}_2(s)$. However, there are drawbacks to this approach:

1. When s is selected uniformly at random from say $\{0,1\}^{1024}$ then $\mathrm{H}_2(s)$ will in general not be perfectly uniform over $\{0,1\}^{512}$. In `YYGen` as defined in Section 3, the input to SHA-512 can be made to be perfectly uniform.

2. Observe that the domain of H_1 is limited to a set of 512-bit strings. Suppose that there is a collision in $\mathrm{H}_1(\mathrm{H}_2(s))$. This implies that there is a value in $\{0,1\}^{512}$ that is not in the range of H_1.

We want $\mathrm{H}_1(\mathrm{H}_2(s))$ to be as uniform as possible over $\{0,1\}^{512}$. So, these drawbacks are an issue.

The second drawback can be heuristically mitigated. Suppose that there exists a hash function SHA-1024 that outputs 1024-bit hashes. Define function f as follows,

$$f(s) = \mathrm{SHA512}(\mathrm{SHA1024}(s))$$

The the upper half of n_c can be set to be $f(s)$ where s is selected from the set of 2048-bit strings. This way, the chances are slim that there exists a value in $\{0,1\}^{512}$ that is not in the range of f. This solution does not address the first drawback, however.

These are only suggestions based on some observations. In an ideal world the open cryptologic research community would work together on a backdoor-resistant RSA key generator.

7 Conclusion

We reviewed the literature on heuristic methods to foil backdoors in RSA key generation. A concrete algorithm was presented that enables a verifier to heuristically verify that a 1024-bit RSA public key has a reduced subliminal channel. It is our hope that this chapter and the corresponding experimental program will influence the adoption of safer RSA key generators. We firmly believe that this work has the potential to mitigate the risks posed by kleptography.

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] Claude Crépeau and Alain Slakmon. Simple backdoors for RSA key generation. In Marc Joye, editor, *Topics in Cryptology CT-RSA, The Cryptographers' Track at the RSA Conference*, pages 403–416. Springer, 2003. Lecture Notes in Computer Science No. 2612.

[3] Y. Desmedt. Abuses in cryptography and how to fight them. In S. Goldwasser, editor, *Advances in Cryptology—Crypto '88*, pages 375–389. Springer-Verlag, 1988. Lecture Notes in Computer Science No. 403.

[4] Yvo Desmedt. Simmons' protocol is not free of subliminal channels. In *Proceedings of the Computer Security Foundations Workshop*, pages 170–175. IEEE Computer Society Press, 1996.

[5] Yvo Desmedt, Claude Goutier, and Samy Bengio. Special uses and abuses of the Fiat-Shamir passport protocol. In Carl Pomerance, editor, *Advances in Cryptology—Crypto '87*, pages 21–39. Springer-Verlag, 1988. Lecture Notes in Computer Science No. 293.

[6] Arjen K. Lenstra. Generating RSA moduli with a predetermined portion. In Kazuo Ohta and Dingyi Pei, editors, *Advances in Cryptology—Asiacrypt '98*, pages 1–10. Springer-Verlag, 1998. Lecture Notes in Computer Science No. 1514.

[7] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[8] National Institute of Standards and Technology (NIST). FIPS Publication 186-2: Digital Signature Standard. January 27, 2000.

[9] National Institute of Standards and Technology (NIST). FIPS Publication 180-2: Announcing the Secure Hash Standard. August 1, 2002.

[10] J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for RSA public-key cryptosystem. *Electronics Letters*, 18(21):905–907, 1982.

[11] RSA Data Security, Inc. *PKCS #1: RSA Cryptography Standard, Version 2.1*, June 2002.

[12] Gustavus J. Simmons. The prisoners' problem and the subliminal channel. In D. Chaum, editor, *Advances in Cryptology—Crypto '83*, pages 51–67. Plenum Press, 1984.

[13] Gustavus J. Simmons. The history of subliminal channels. *IEEE Journal on Selected Areas in Communication*, 16(4):452–462, May 1998.

[14] Adam Young. Mitigating insider threats to RSA key generation. *CryptoBytes*, 7(1):1–15, 2004.

[15] Adam Young and Moti Yung. A space efficient backdoor in RSA and its applications. In Bart Preneel and Stafford E. Tavares, editors, *Selected Areas in Cryptography—SAC '05*, pages 128–143. Springer, 2005. Lecture Notes in Computer Science No. 3897.

[16] Adam L. Young and Moti M. Yung. The dark side of black-box cryptography, or: Should we trust capstone? In Neal Koblitz, editor, *Advances in Cryptology—Crypto '96*, pages 89–103. Springer-Verlag, 1996. Lecture Notes in Computer Science No. 1109.

[17] Adam L. Young and Moti M. Yung. Kleptography: Using cryptography against cryptography. In Walter Fumy, editor, *Advances in Cryptology—Eurocrypt '97*, pages 62–74. Springer-Verlag, 1997. Lecture Notes in Computer Science No. 1233.

[18] Adam L. Young and Moti M. Yung. The prevalence of kleptographic attacks on discrete-log based cryptosystems. In Burton S. Kaliski, editor, *Advances in Cryptology—Crypto '97*, pages 264–276. Springer-Verlag, 1997. Lecture Notes in Computer Science No. 1294.