Bug Attacks

Eli Biham[1], Yaniv Carmeli[1], and Adi Shamir[2]

[1] Computer Science Department,
Technion - Israel Institute of Technology,
Haifa 32000, Israel
{biham,yanivca}@cs.technion.ac.il
http://www.cs.technion.ac.il/∼{biham,yanivca}

[2] Computer Science Department,
The Weizmann Institute of Science,
Rehovot 76100, Israel
adi.shamir@weizmann.ac.il

Abstract. In this paper we present a new kind of cryptanalytic attack which utilizes bugs in the hardware implementation of computer instructions. The best known example of such a bug is the Intel division bug, which resulted in slightly inaccurate results for extremely rare inputs. Whereas in most applications such bugs can be viewed as a minor nuisance, we show that in the case of RSA (even when protected by OAEP), Pohlig-Hellman, elliptic curve cryptography, and several other schemes, such bugs can be a security disaster: Decrypting ciphertexts on *any* computer which multiplies *even one pair of numbers* incorrectly can lead to full leakage of the secret key, sometimes with a single well-chosen ciphertext.

Keywords: Bug attack, Fault attack, RSA, Pohlig-Hellman, ECC.

1 Introduction

With the increasing word size and sophisticated optimizations of multiplication units in modern microprocessors, it becomes increasingly likely that they contain some undetected bugs. This was demonstrated by the accidental discovery of the Pentium division bug in the mid 1990's, by the less famous Intel 80286 `popf` bug (that set and then cleared the interrupt-enable bit during execution of the very simple `popf` instruction, when no change in the bit was necessary), by the recent discovery of a bug in the Intel Core 2 memory management unit (which allow memory corruptions outside of the range of permitted writing for a process), etc.

In this paper we show that if some intelligence organization discovers (or secretly plants) even one pair of single-word integers a and b whose product is computed incorrectly (even in a single low order bit) by a popular microprocessor, then any key in any RSA-based security program running on any one of the millions of PC's that contain this microprocessor can be easily broken, unless appropriate countermeasures are taken. In some cases, the full key can be retrieved with a single chosen ciphertext, while in other cases (such as RSA protected by the popular OAEP technique), a larger number of ciphertexts is required. The attack is also applicable to other cryptographic schemes which are based on exponentiation modulo a prime or on point multiplication in elliptic curves, and thus almost all the presently deployed public key schemes are vulnerable to such an attack.

The new attack, which we call a *Bug Attack*, is related to the notion of fault attacks discovered by Boneh, Demillo and Lipton in 1996 [4], but seems to be much more dangerous in its implications. The original fault attack concentrated on soft errors that yield random results when induced at a particular point of time by the attacker (latent faults were briefly mentioned, but were never studied). They require physical possession of the computing device by the attacker, and the deliberate injection of a transient fault by operating this device in an unusual way (e.g., in a microwave oven, at high temperature, with high frequency clock, or with a sudden spike in the power supply). Such attacks are feasible against smart cards, but are much harder to carry out against PC's. In the new bug attack, the target PC's can be located at secure locations half a world away, and millions of PC's can be attacked simultaneously over the Internet, without having to manipulate the operating environment of each one of them individually. Unlike the case of fault attacks, in bug attacks the error is deterministic, and is triggered whenever a particular computation is carried out; the attacker cannot choose the timing or nature of the error, except by choosing the inputs of the computation.

Since the design of modern microprocessors is usually kept as a trade secret, there is no efficient method for the user to verify that a single multiplication bug does not exist. For example, there are 2^{128} pairs of inputs in a 64×64 bit multiplier, so we cannot try them all by exhaustive search. We can even expect that most of the 2^{128} pairs of inputs will never be multiplied on any processor. Even if we assume that Intel had learned its lesson and meticulously verified the correctness of its multipliers, there are many smaller manufacturers of microprocessors who may be less careful with their design, and less careful in testing the quality of the chips they produce. In addition, many PC's are sold with overclocked processors which are more likely to err when performing complex instructions such as 64-bit integer multiplication. The problem is not limited to microprocessors: Many cellular telephones are running RSA or elliptic curve computations on signal processors made by TI and others, FPGA or ASIC devices can embed in their design flawed multipliers from popular libraries of standard cell designs, and many security programs use optimized "bignum packages" written by others without being able to fully verify their correctness.

In addition to such innocent bugs, there is the issue of intentionally tampered hardware, which is a major security problem. In February 2005 the matter was addressed in a US Department of Defense (DoD) report [17], which warned about the risks of importing hardware from foreign countries to the US. Recently the F.B.I. reported that 3,500 counterfeit Cisco network components were discovered in the US, and some of them had even found their way into US military and government facilities [9]. Although in this case Cisco did not find any evidence of malicious modifications, it certainly demonstrates the feasibility of such a scenario. In [7], the open source design of the Leon3 processor was changed to exemplify that a hardware backdoor can be introduced into processors. The change (which affected only 0.05% of the logic gates) was shown to allow attackers to load a malicious firmware and take full control of the attacked machine. Even commercially sold bug-free processors can be made buggy by anyone along the supply chain who modifies their firmware via their built-in bug-fixing mechanisms.

What we show in this paper is that the innocent or intentional introduction of any bug into the multiplier of any processor (even when it affects only two specific inputs whose product contains a single erroneous low-order bit) can lead to a major security disaster, which can be secretly exploited in an essentially undetectable way by a sophisticated intelligence organization. Even though we are not aware of any such attacks being carried out in practice, hardware manufacturers and security experts should be aware of this possibility, and use appropriate countermeasures.

In this paper we present bug attacks against several widely deployed cryptosystems (such as Pohlig-Hellman [11], RSA [13], elliptic curve schemes, and some symmetric primitives), and against several implementations of these schemes. For all the discussed schemes, we show that the secret exponent can be retrieved by a chosen ciphertext attack, and in the case of Pohlig-Hellman, the secret exponent can also be retrieved by a chosen plaintext attack. In the case of RSA, we show that if decryption is performed using the Chinese remainder theorem (CRT) [10, Note 14.70] the public modulus n can be factored using a single chosen ciphertext. A particularly interesting observation is that even though RSA-OAEP [1] was designed to prevent chosen ciphertext attacks, we can actually use this protective mechanism as part of our bug attack in order to learn whether a bug was or was not encountered during the exponentiation process. This demonstrates that in spite of the similarity between bug attacks and fault attacks, their countermeasures can be very different. For example, just stopping an erroneous computation or recomputing the result with a different exponentiation algorithm may protect the scheme against fault attacks, but will leak the full key via a bug attack.

This paper is organized as follows: Section 2 gives an overview of the methods we use in most of our attacks, and describes the two most commonly used implementations of modular exponentiations: the left-to-right (LTOR) and right-to-left (RTOL) exponentiation algorithms. Section 3 presents the simplest bug attack on RSA when decryption is performed using the Chinese remainder theorem (CRT), using a single chosen ciphertext. Section 4 presents attacks on several cryptosystems when exponentiations are computed using the LTOR algorithm, and Section 5 presents attacks on the same schemes when the exponentiations are computed using the RTOL algorithm. In Section 6 we discuss bug attacks on elliptic curve schemes and some symmetric primitives. Section 7 summarizes the contributions of this paper, and presents the time and data complexities of all our attacks. Finally, Appendix A provides descriptions of the cryptosystems discussed in this paper.

2 Overview of Our Methods and Notations

We present several attacks which use multiplication bugs. We concentrate on these operations since multiplication and division are typically the most complex operations, their implementations are most aggressively optimized, and therefore bugs are more likely to exist in them than in simple operations like addition or XOR, and are less likely to be discovered by the manufacturers.

2.1 Multiplication of Big Numbers

In cryptography, we are often required to perform arithmetic operations on big numbers, which must be represented using more than a single 32-bit or 64-bit word. Arithmetic operations on such values must be broken down into arithmetic operations on the different words which comprise them. For example, when multiplying two very long integers x and y, each represented by ten words, each of the ten words of x is multiplied by each of the ten words of y, in some order, and the results are then summed up to the appropriate words of the product. If x contains a in the sense that one of the ten words of x is a, y contains b, and the processor produces an incorrect result when a and b are multiplied, then the result of multiplying $x \cdot y$ on that processor will typically be incorrect (unless there are multiple errors that exactly cancel each other during the computation, which is very unlikely when the other words in x and y are randomly chosen).

2.2 Notations

We use the notation $x \cdot y$ to denote the result of multiplying x by y on a bug-free processor, and $x \odot y$ to denote the result of the same computation when performed on a faulty processor. Similarly, the notation x^l denotes the value of x to the power l as computed on a bug-free processor, and $x^{\langle l \rangle}$ denotes the value of x to the power l as computed by a particular algorithm on a faulty processor (See Section 2.5 for details of popular exponentiation algorithms). Since we assume that faults are extremely rare, for most inputs we expect the result of the computation to be the same on both the faulty and the bug-free processors, and in these cases we use the notations $x \cdot y$ and x^l, even when referring to computations done on the faulty processor.

2.3 Methods

Our attacks request the decryptions of ciphertexts which may or may not invoke the execution of the faulty multiplications, depending on the bits of the secret exponent d. The results of those decryptions are used to retrieve the bits of the secret exponent d. We develop two methods for creating the conditions under which the buggy instructions are executed. The first method chooses a ciphertext C, such that an intermediate value x during the decryption process contains both a and b. If x is squared, then we expect that $x^2 \neq x^{\langle 2 \rangle}$, and thus the result of the entire decryption process is also expected to be incorrect. If x is multiplied by a different value y, which contains neither a nor b, then we expect that $x \cdot y = x \odot y$, and the decryption result is expected to be correct. The second method chooses C such that during decryption one intermediate value x

```
LTOR Exponentiation

$z \leftarrow 1$
For $k = \log n$ down to 0
   If $d_k = 1$ then $z \leftarrow z^2 \cdot x \bmod n$
   Otherwise, $z \leftarrow z^2 \bmod n$
Output $z$
```

```
RTOL Exponentiation

$y \leftarrow x; z \leftarrow 1$
For $k = 0$ to $\log n$
   If $d_k = 1$ then $z \leftarrow z \cdot y \bmod n$
   $y \leftarrow y^2 \bmod n$
Output $z$
```

Fig. 1. The Two Basic Exponentiation Algorithms

contains a, while another value y contains b. If x and y are multiplied then it is expected that $x \cdot y \neq x \odot y$, and the result of decryption on the faulty processor is expected to be incorrect. If x and y are not multiplied by the decryption algorithm, we expect the decryption to be correct.

2.4 Complexity Analysis

Let w be the length (in bits) of the words of the processor. When analyzing complexities of our attacks throughout this paper we assume that numbers (both exponentiated values and exponents) are 1024-bit long, and that $w = 32$ (in the summary of the paper we also quote the complexities for $w = 64$). The standard representation of 1024-bit long numbers requires $\lceil 2^{10}/w \rceil$ words. Given a random 1024-bit value x, and a w-bit value a, the probability that x contains a (in any of its $2^{10}/w$ words) is about $2^{-w}2^{10}/w$. For $w = 32$ this probability is 2^{-27}, and for $w = 64$ it is 2^{-60}. Given two w-bit values a and b, the probability that x contains both a and b is about $\left(2^{-w}2^{10}/w\right)^2$. For $w = 32$ this probability is about 2^{-54}, and for $w = 64$ it is about 2^{-120}.

2.5 Exponentiation Algorithms

Given a value x and a secret exponent $d = d_{\log n} d_{\log n - 1} \ldots d_1 d_0$, the exponentiation $x \mapsto x^d \bmod n$ can be efficiently computed by several exponentiation algorithms [10, Chapter 14.6]. In this paper we present attacks against implementations that use the two basic exponentiation algorithms, LTOR (left-to-right) and RTOL (right-to-left), described in Figure 1. Our techniques can be easily adapted to attack implementations that use other exponentiation algorithms such as the sliding window algorithm, the k-ary exponentiation algorithm, etc.

2.6 Remarks

The following remarks apply to most of the attacks presented in this paper.

1. Microprocessors usually perform different sequences of microcode instructions when computing $a \cdot b$ and $b \cdot a$, and thus the bug is not expected to be

symmetric: for $a \cdot b$ the processor may give an incorrect result, while for $b \cdot a$ the result is correct. Therefore, the correctness of the result of multiplying two big numbers x and y, where x contains a and y contains b, depends on whether the implementation of $x \cdot y$ multiplies $a \cdot b$ or $b \cdot a$. We assume that such implementation details are known to the attacker.

2. Given a value w, the number of bits in the binary representation of w is $\lfloor \log_2 w \rfloor + 1$ (the indices of the bits of w are $0, \ldots, \lfloor \log w \rfloor$, where 0 is the index of the least significant bit, and $\lfloor \log w \rfloor$ is the index of the most significant bit). Throughout this paper we use $\log w$ (without the floor operator) as a shorthand for the index of the most significant bit of w.
3. It may be the case that more than one pair of buggy inputs $a \cdot b$ exist. In such cases, if γ multiplication bugs are known to the attacker, the complexities of some of the attacks we present can be decreased. In attacks where the attacker can control only one of the operands of the multiplication, and the other operand is expected to appear randomly, the time complexity can be decreased by a factor of $\min(\gamma, \lfloor \log n/w \rfloor)$. If some of the buggy pairs of operands share the same value for one of the operands, this factor can even get better (but it cannot be higher than γ). In attacks where both operands are expected to appear randomly, the time complexity can be decreased by a factor of γ. Note that symmetric bugs, where both the results of $a \cdot b$ and $b \cdot a$ are incorrect, are counted as two bugs.
4. If both operands of the buggy instruction are equal (i.e., $a = b$), the complexity of some of our attacks can be greatly reduced, while other attacks become impossible. The former case happens when attacks rely on faults in the squaring of values X, where X happens by chance to contain both a and b. In this case only one word (a) needs to appear in X, which makes the probability of this event much higher. On the other hand, attacks which use the existence of a bug in order to decide whether x and y were squared or multiplied together become impossible. When the attack requires that x contains a and that y contains b, our ability to distinguish between these cases depends on whether $a = b$.

3 Breaking CRT-RSA with One Chosen Ciphertext

We now describe a simple attack on RSA implementations in which decryptions are performed using the Chinese remainder theorem (CRT). Let $n = pq$ be the public modulus of RSA, where p and q are large primes, and assume without loss of generality that $p < q$. Knowing the target's public key n (but not its secret factors p and q), the attacker can easily compute a half size integer which is guaranteed to be between the two secret factors p and q of n. For example, $\lfloor \sqrt{n} \rfloor$ always satisfies $p \leq \lfloor \sqrt{n} \rfloor < q$, and any integer close to $\sqrt{n}$ is also likely to satisfy this condition. The attacker now chooses a ciphertext C which is the closest integer to $\sqrt{n}$, such that both a and b appear as low order words in C, and submits this "poisonous input" to the target PC.

The first step in the CRT-RSA computation is to reduce the input C modulo p and modulo q. Due to its choice, $C_p = C \bmod p$ is randomized modulo the smaller factor p, but $C_q = C \bmod q = C$ remains unchanged modulo the larger factor q. The next step in RSA-CRT is always to square the reduced inputs C_p and C_q, respectively. Since a and b are unlikely to remain in C_p, the computation mod p is likely to be correct. However, mod q the squaring operation will contain a step in which the word a is multiplied by the word b, and by our assumption the result will be incorrect. Assuming that the rest of the two computations mod p and q will be correct, the final result of the two exponentiations will be combined into a single output $\hat{M}$ which is likely to be correct mod p, but incorrect mod q. The attacker can then finish off his attack in the same way as the original fault attack, by computing the greatest common divisor (gcd) of n and $\hat{M}^e - C$, where e is the public exponent of the attacked RSA key. This gcd is the secret factor p of n.

Note that if such C cannot be found, then $q - p < 2^{2w}$. In this case, n can be easily factored by other methods (e.g., Fermat's factorization method, which will factor n in 2^w time without any calls to the decryption oracle).

4 Bug Attacks on LTOR Exponentiations

In this section we present bug attacks against several cryptosystems, where exponentiations are performed using the LTOR exponentiation algorithm. We first present chosen plaintext (or chosen ciphertext) attacks against the Pohlig-Hellman scheme, then present chosen ciphertext attacks against RSA, and finally discuss how to adapt our attacks on RSA to the case of RSA-OAEP.

4.1 Bug Attacks on Pohlig-Hellman

The Pohlig-Hellman cipher uses two secret exponents e and d: the former is used for encryption, and the latter for decryption. Given one of the secret exponents, the other can be computed by $d \equiv e^{-1} \pmod{p-1}$. We discuss adaptive and non-adaptive chosen ciphertext attacks which retrieve the bits of the decryption exponent d; similar chosen plaintext attacks can retrieve the encryption exponent e.

We start by presenting a simple adaptive attack, which demonstrates the basic idea of our technique. We later improve this attack with additional ideas.

4.1.1 Basic Adaptive Chosen Ciphertext

In this section, an attack which requires the decryption of $2 \log p$ chosen ciphertexts is presented. The attack retrieves the bits of the secret exponent one at a time, from $d_{\log p}$ to d_1 (d_0 is known to be one, as d is odd). Therefore, when the search for d_i is performed, we can assume that the bits $d_{i+1}, \ldots, d_{\log p}$ are already known.

The attack works as follows:

1. Choose a value X which contains the words a and b.
2. For $i = \log p$ down to 1 do

   (a) Denote the value of the known bits of d by $d' = \sum_{k=i+1}^{\log p} 2^{k-(i+1)} d_k$.
   (b) Compute $C = X^{1/d'} \bmod p$.
   (c) Ask for the decryption $\hat{M} = C^{\langle d \rangle} \bmod p$ on the faulty processor.
   (d) Obtain the correct decryption $M = C^d \bmod p$.
   (e) If $M = \hat{M}$ conclude that $d_i = 1$, otherwise conclude that $d_i = 0$.
3. Set $d_0 = 1$.

The attack is based on the following observations. Since p is a known prime, the attacker can compute arbitrary roots modulo p. During the i'th iteration of the attack, the value of C is chosen such that when it is exponentiated to power d with LTOR, the intermediate value of the variable z after $\log p - i$ iterations is X. The next operation of the LTOR algorithm is either squaring z, or multiplying it by C, depending on the value of d_i. Since the intermediate value $z = X$ contains both a and b, we expect to get an incorrect decryption if z is squared (i.e., when $d_i = 0$), and a correct decryption if z is first multiplied by C (i.e., when $d_i = 1$).

Note that the bug-free decryption in Step 2d may be obtained on the same buggy microprocessor by using the multiplicative property of modular exponentiation. The attacker may request the decryption M' of $C' = C^3 \bmod p$ (or any other power of C which is not expected to cause the execution of the faulty instructions), and then check whether $\hat{M}^3 \equiv M' \pmod{p}$ to learn if an error had occurred. Thus, no calls to a bug-free decryption device that uses the same secret key (which is usually unavailable) is required. In fact, since the same value of X is used for each of the iterations, the correct decryption M can be computed from the value of the correct decryption in the previous iteration as: $M = \bar{M}^{d'/\bar{d}'} \bmod n$, where $\bar{M}$ and $\bar{d}'$ are the values of the corresponding variables in the previous iteration. Therefore, no additional decryption requests (beyond the first one) are needed in order to obtain all the correct decryption results throughout the attack.

The attack requires buggy decryption of $\log p + 1$ chosen ciphertexts to retrieve d, or buggy encryption of $\log p + 1$ chosen plaintexts to retrieve e. Each one of these values makes it easy to compute the other value since p is a known prime.

4.1.2 Improved Adaptive Chosen Ciphertext Attack

We observe that X can be selected such that both X and $X^{\langle 2 \rangle}$ contain a and b. A further improvement uses X's which contain a and b, such that when X is squared m times repeatedly on a faulty processor (for some $m > 0$), all the values $X^{\langle 2^j \rangle}$ contain a and b. Using such X, we can improve the expected complexity of the attack by a factor of $\alpha = 2 - 2^{-m}$. Further details on this improved attack will be presented in the full version of this paper.

4.1.3 Chosen Ciphertext Attack

The (non-adaptive) chosen ciphertext attack presented later in Section 4.2.2 is also applicable in the case of Pohlig-Hellman. The attack requires decryption of 2^{28} ciphertexts to retrieve the secret exponent d (the attack on RSA requires 2^{27} ciphertexts, but in the case of Pohlig-Hellman an additional decryption is required for each buggy decryption, in order to verify the correctness of the decryption). As in the previous attacks on Pohlig-Hellman, a similar chosen plaintext attack can retrieve the secret exponent e.

4.2 Bug Attacks on RSA

We describe several chosen ciphertext attacks on RSA, where the attacked implementation performs decryptions without using CRT. Instead, we assume that the decryption of a ciphertext C is performed by computing $C^d \bmod n$ using LTOR (where d is the secret exponent of RSA). We assume that the public exponent e and the public modulus n are known. The main difference between the case of RSA and the case of Pohlig-Hellman is that there is no known efficient algorithm to compute roots modulo a composite n, when the factorization of n is unknown.

Unlike the case of Pohlig-Hellman, in the case of RSA checking whether the decrypted message $\hat{M}$ is the correct decryption of a chosen ciphertext C can be easily done by checking whether $\hat{M}^e \equiv C \pmod{n}$. Thus, there is no need to request the decryptions of additional messages for this purpose.

4.2.1 Adaptive Chosen Ciphertext Attack

We describe an adaptive chosen ciphertext attack which requires the decryption of $\log n$ chosen ciphertexts by the target computer. The generation of each of the ciphertexts requires 2^{27} time on the attacker's (bug-free) computer, and thus the total time complexity of the attack is about 2^{37}.

Description of the attack:

1. For $i = \log n$ down to 1 do
   (a) Denote the value of the known bits of d by $d' = \sum_{k=i+1}^{\log n} 2^{k-(i+1)} d_k$.
   (b) Repeatedly choose random values C which contain b, until $C^{d'} \bmod n$ contains a.
   (c) Ask for the decryption $\hat{M} = C^{\langle d \rangle} \bmod n$ using the faulty processor.
   (d) Compute $\hat{C} = \hat{M}^e \bmod n$.
   (e) If $\hat{C} = C$ conclude that $d_i = 0$, otherwise conclude that $d_i = 1$.
2. Set $d_0 = 1$.

The attack is similar to the basic attack presented in Section 4.1.1, except that here only the word a is contained in the intermediate value of the exponentiation. The word b is contained in the ciphertext C, and therefore the roles of the correct and incorrect results are exchanged between $d_i = 0$ and $d_i = 1$.

During the execution of the LTOR algorithm, the intermediate value of the variable z after $\log n - i$ iterations contains a (due to the selection of C in Step 1b of the attack). If $d_i = 0$ then z is squared, and no errors in the computation are expected to occur, leading to $\hat{C} = C$ in Step 1e. If $d_i = 1$, then z is multiplied by C, which contains the word b, and due to the bug, the result of the exponentiation is expected to be incorrect, leading to $\hat{C} \neq C$ in Step 1e.

As explained in Section 2, the probability that the random number $C^{d'} \bmod n$ contains somewhere along it the word a is 2^{-27} (for our standard parameters).

Therefore, Step 1b takes an average time of 2^{27} exponentiations on the attacker's computer.

4.2.2 Chosen Ciphertext Attack

The previous adaptive attack on exponentiations using LTOR is the basis for the following non-adaptive chosen ciphertext attack. The attack requests the decryption of 2^{29} chosen ciphertexts, all of which contain the word b. It is expected that for every $0 \leq i \leq \log n$, there are about four ciphertexts for which the intermediate value of z after i rounds of the exponentiation algorithm contains the word a. The value of d_i can be determined by the correctness of the decryption of those ciphertexts, using considerations similar to the ones used in the attack of Section 4.2.1. If for some i there are no ciphertexts C_j for which $X_j = C_j^{d'} \bmod n$ contains a, there is no choice but to continue the attack recursively for both $d_i = 0$ and $d_i = 1$. However, when the wrong value is chosen, a contradiction may be encountered before retrieving the rest of the bits (i.e., more than one ciphertext C_j for which X_j contains a is found, and the decryption of some, but not all, of them is incorrect). By using standard results from the theory of branching processes, 2^{29} ciphertexts suffice to ensure that recursive calls which represent wrong bit values are quickly aborted.
Here are some details of this attack:

1. Choose 2^{29} random ciphertexts C_j $(1 \leq j \leq 2^{29})$ containing the word b, and ask for their decryptions $\hat{M}_j$ using the faulty processor.
2. For $i = \log n$ down to 1 do
   (a) Denote the value of the known bits of d by $d' = \sum_{k=i+1}^{\log n} 2^{k-(i+1)} d_k$.
   (b) For each ciphertext C_j compute $X_j = C_j^{d'} \bmod n$.
   (c) Consider all ciphertexts C_j such that X_j contains a:
      i. If for all such ciphertexts C_j it holds that $\hat{M}_j^e \bmod n = C_j$ then set $d_i = 0$.
      ii. If for all such ciphertexts C_j it holds that $\hat{M}_j^e \bmod n \neq C_j$ then set $d_i = 1$.
      iii. If there are no such ciphertexts try the rest of the attack for both $d_i = 0$ and $d_i = 1$.
      iv. If for some of these ciphertexts C_j, $\hat{M}_j^e \bmod n = C_j$ and for others $\hat{M}_j^e \bmod n \neq C_j$ (i.e., a previously set value of one of the bits is wrong) then backtrack.
3. Set $d_0 = 1$.

The data complexity may be increased in order to decrease the probability of not having ciphertexts C_j such that X_j contains a. Alternatively, it may be decreased, at the expense of more recursive guesses (Step 2(c)iii), with increased time complexity. If for every i there exists a j such that $C_j^{d'}$ contains b, the time complexity is equal to the data complexity (i.e., 2^{29}).

4.2.3 Known Plaintext Attack

The chosen ciphertext attack from Section 4.2.2 can be easily transformed into a known plaintext attack which requires 2^{56} known plaintexts. Among the 2^{56} plaintexts, only 2^{29} are expected to contain b. We can discard all the plaintexts which do not contain b, and use the rest as inputs for the attack described in Section 4.2.2.

Note that the known plaintexts must be the result of decrypting the corresponding ciphertexts on the faulty processor. The attack will not work if the given plaintext-ciphertext pairs are obtained by encrypting plaintexts (either on the attacker's computer or on the target computer).

4.3 Bug Attacks on OAEP

Since RSA has many mathematical properties such as multiplicativity, it is often used in modes of operation which protect it against attacks based on these properties. The most popular mode is OAEP [1], which is provably secure. We show here that although OAEP protects against "standard" attacks on RSA, it provides only limited protection against bug attacks, since it was not designed to deal with errors during the computation.

OAEP adds randomness and redundancy to messages before encrypting them with RSA, and rejects ciphertexts which do not display the expected redundancy when decrypted. Random ciphertexts are not expected to display such a redundancy, and are likely to be rejected by the receiver with overwhelming probability. To choose valid ciphertexts with certain desired characteristics, we choose random plaintexts and encrypt them using proper OAEP padding, until we get a ciphertext that has the desired structure by chance (since OAEP is a randomized cipher, we can also try to encrypt the same message with different random values, and thus can control the result of the decryption). Our main observation is that the structure we need in our attack (such as the existence of a certain word in the ciphertext) has a relatively high probability regardless of how much redundancy is added to the plaintext by OAEP, and the knowledge that a correctly constructed ciphertext was rejected suffices to conclude that some computational error occurred. We are thus exploiting the OAEP countermeasure itself in order to mount the new bug attack!

The attacks we present on RSA-OAEP are very similar to the attacks on RSA from Section 4.2, with some minor modifications. The same attacks are also applicable to OAEP+ [16].

4.3.1 Adaptive Chosen Ciphertext Attack

Unlike the attack of Section 4.2.1, OAEP stops us from choosing ciphertexts C which contain b, and thus in Step 1b we must choose random messages (on our own computer) until b "appears" in C at random. As explained in Section 2.4, the probability that this happens and in addition $C^{d'} \bmod n$ contains a is 2^{-54}. As mentioned above, computation errors are identified in Step 1e of the attack on OAEP by the mere rejection of the ciphertext, and there is no need to know the actual value which was rejected. The attack requires the decryption of $\log n$ chosen ciphertexts, and thus its total time complexity for 1024-bit n's is 2^{64}.

4.3.2 Chosen Ciphertext Attack

The (non-adaptive) chosen ciphertext attack on RSA from Section 4.2.2 can also be used in the case of OAEP. For a random message, the probability that the ciphertext contains b is 2^{-27}. In order to find 2^{29} messages with a ciphertext which contains b (as required by the attack), we have to try about 2^{56} random messages. Therefore, the attack requires the decryption of 2^{29} chosen ciphertexts, plus 2^{56} pre-computation time on the attacker's own computer. Once the decryptions of the chosen ciphertexts are available, the key can be retrieved in 2^{29} additional time.

5 Bug Attacks on RTOL Exponentiations

In this section we present attacks against Pohlig-Hellman, RSA, and RSA-OAEP, where exponentiations are performed using the RTOL exponentiation algorithm. In RTOL, the value of the variable y is squared in every iteration of the exponentiation algorithm, regardless of the bits of the secret exponent. Any error introduced into the value of y undergoes the squaring transformation in every subsequent iteration, and is propagated to the value of z if and only if the corresponding bit of the exponent is set. Consequently, every set bit in the binary representation of the exponent introduces a different error into the value of z, while zero bits do not introduce any errors. This allows us to mount efficient non-adaptive attacks, and to retrieve more than one bit from each chosen ciphertext, as described in the attacks presented in this section.

5.1 Bug Attacks on Pohlig-Hellman

We present a chosen ciphertext attack against Pohlig-Hellman, where exponentiations are performed using RTOL. The attack is aimed at retrieving the bits of the secret exponent d. As in Section 4.1, an identical chosen plaintext attack can retrieve the bits of the secret exponent e.

5.1.1 Chosen Ciphertext Attack

We present a (non-adaptive) chosen ciphertext attack which retrieves the secret key when the exponentiation is performed using RTOL. Let X be a value which contains the words a and b, and let $\beta = X^2/X^{\langle 2\rangle}$. Unlike the improved attack on Pohlig-Hellman of Section 4.1.2, it does not help if $X^{\langle 2\rangle}$ also contains a and b (on the contrary, it makes the analysis slightly more complicated). Each chosen ciphertext is used to retrieve r bits of the secret exponent d, where r is a parameter of the attack. The reader is advised to consider first the simplest case of $r = 1$.

The attack is carried out using the following steps:

1. For $i = \log p - (\log p \bmod r)$ down to 0 step $-r$
   (a) Compute $C = X^{1/2^{i-1}} \bmod p$.
   (b) Denote the value of the known bits of d by $d' = \sum_{k=i+r}^{\log p} 2^{k-(i+r)} d_k$.
   (c) Ask for the decryption $\hat{M} = C^{\langle d \rangle} \bmod p$ on the faulty processor.
   (d) Obtain the correct decryption $M = C^d \bmod p$.
   (e) Find an r-bit value u such that $M/\hat{M} = \beta^{2^r d' + u} \bmod p$ $(0 \le u < 2^r)$.
   (f) Denote the bits of u by $u_{r-1}u_{r-2}\ldots u_1 u_0$.
   (g) Conclude that $d_{i+k} = u_k, \quad \forall\, 0 \le k < r$.

Consider the decryption of C in Step 1c, for some i. Exponentiation by the RTOL algorithm sets $y = C$, and squares y repeatedly. After squaring it $i-1$ times, the value of y becomes X, which contains both a and b. When y is squared again, a multiplicative error factor of β is introduced into its computed value (compared to its bug-free value). If $d_i = 1$ then z is multiplied by y, and thus the same multiplicative error factor of β is also propagated into the value of z. After the next squaring of y, it contains an error factor of β^2, which is propagated into the value of z when $d_{i+1} = 1$. In each additional iteration of the exponentiation the previous error in y is squared, and the error affects the result if and only if the corresponding bit of d is set. At the end of the exponentiation, the error factor in the final result is:

$$\frac{M}{\hat{M}} \equiv \prod_{k=i}^{\log p} \left(\beta^{2^{k-i}}\right)^{d_k} \equiv \beta^{\sum_{k=i}^{\log p} 2^{k-i} d_k} \pmod{p}.$$

Since only r bits of the exponent are unknown, they can be easily retrieved by performing $2^r - 1$ modular multiplications.

As in the attacks of Section 4.1, all the error-free decryption queries in Step 1c can be replaced by the decryption of one additional ciphertext on the faulty processor: The attacker can request the decryption M^3 of $C^3 \bmod p$ (or any other power of C which is not expected to cause a decryption error), and then in Step 1e can find an r-bit value u such that

$$\frac{M^3}{\hat{M}^3} \equiv \left[\prod_{k=i}^{\log p} \left(\beta^{2^{k-i}}\right)^{d_k}\right]^3 \equiv \beta^{3\left(2^r d' + u\right)} \pmod{p}.$$

The attack requires $2\lceil(\log p + 1)/r\rceil$ decryptions of chosen ciphertexts, and all of them can be pre-computed by $\log p$ modular square roots (Step 1a of the attack). Once the decryptions are available, each execution of Step 1e finds r bits of d using $2^r - 1$ multiplications, which is equivalent to about $2^r/\log p$ modular exponentiations. Since Step 1e is executed $\lceil(\log p + 1)/r\rceil$ times, the total time complexity is about $2^r/r$. For small values of r this time complexity is negligible compared to the time of the pre-computation. For $r \ge 12$, however, this computation takes longer, and there is a tradeoff between the time complexity and the data complexity.

5.2 Bug Attacks on RSA

Unlike the case of Pohlig-Hellman, there is no known efficient algorithm for extracting roots modulo a composite n with unknown factors. The chosen ciphertext attack presented in this section circumvents this problem by choosing random ciphertexts until a suitable ciphertext is found.

5.2.1 Chosen Ciphertext Attack

The attack in this case is similar to the attack on RTOL modulo a prime p (Section 5.1.1), except for some necessary adaptations to the case of RSA. The attack requires a pre-computation to find a value X which contains both a and b, and such that all the values $X^{1/2^{i-1}}$ for $1 \leq i \leq \log n$ are known (Step 2 in the following attack). The parameter r represents the number of bits retrieved in each iteration.

The detailed attack is as follows:

1. Choose a random ciphertext C_0, and let $t = 0$.
2. While $t \leq \log n$ or C_t does not contain both a and b do:
   (a) $t = t + 1$.
   (b) Compute $C_t = C_{t-1}^2 \bmod n$.
3. Let $X = C_t$ and let $X^{\langle 2 \rangle}$ be the result of squaring X on a faulty processor.
4. Let $\beta = X^2 / X^{\langle 2 \rangle} \bmod n$.
5. For $i = \log n - (\log n \bmod r)$ down to 0 step $-r$
   (a) Ask for the decryption $\hat{M}$ of $C = C_{t-i}$ using the faulty processor, $M = C_{t-i}^{\langle d \rangle} \bmod p$.
   (b) Denote the value of the known bits of d by $d' = \sum_{k=i+r}^{\log n} 2^{k-(i+1)} d_k$.
   (c) Compute $\hat{C} = \hat{M}^e \bmod n$.
   (d) Find an r-bit value u such that $C/\hat{C} \equiv \left(\beta^{2^r d' + u}\right)^e \pmod{n}$.
   (e) Denote the bits of u by $u_{r-1} u_{r-2} \ldots u_1 u_0$.
   (f) Conclude that $d_{i+k} = u_k, \quad \forall\ 0 \leq k < r$.

A random ciphertext contains a and b with probability 2^{-54}, and therefore the pre-computation of Step 2 is expected to take time corresponding to 2^{54} modular multiplications (which is equivalent to 2^{44} modular exponentiations when $\log n = 1024$). In each iteration of the attack, r bits are retrieved by performing $2^r - 1$ modular multiplications, which are equivalent to about $(2^r - 1)/\log n$ modular exponentiations. Thus, once the decrypted ciphertexts are available, the attack requires a time equivalent to about

$$\left\lceil \frac{\log n}{r} \right\rceil \frac{2^r - 1}{\log n} \approx \frac{2^r - 1}{r}$$

modular multiplications. As in the attack of Section 5.1, this attack requires $\lceil \log n / r \rceil$ decryptions of pre-computed chosen ciphertexts. Step 5d finds r bits of the secret exponent d using $2^r - 1$ multiplications, and thus (as in the attack from Section 5.1.1) for large values of r there is a tradeoff between the time complexity and the data complexity.

5.3 Bug Attacks on OAEP Implementations That Use RTOL

5.3.1 Adaptive Chosen Ciphertext Attack

We present an adaptive chosen ciphertext attack for the case of RSA-OAEP when exponentiations are performed using RTOL. The presented attack resembles the attack from Section 4.3, but it identifies the bits of d starting from the *least* significant bit.

The description of the attack is as follows:

1. Set $d_0 = 1$.
2. For $i = 1$ to $\log n$
   (a) Denote the value of the known bits of d by $d' = \sum_{k=0}^{i-1} 2^k d_k \bmod n$.
   (b) Repeatedly encrypt random messages M until $C = E(M) = (\text{OAEP}(M))^e$ satisfies that $C^{2^i} \bmod n$ contains a and $C^{d'} \bmod n$ contains b.
   (c) Ask for the decryption of C using the faulty processor.
   (d) If the decryption succeeds conclude that $d_i = 0$, otherwise conclude that $d_i = 1$.

After i iterations of the decryption exponentiation algorithm, the value of the variable z is $C^{d'} \bmod n$, and the value of the variable y is $C^{2^i} \bmod n$. The ciphertext C is chosen such that one of these values contains a and the other contains b. Therefore, if these values are multiplied ($d_i = 1$), then the result of the decryption is expected to be wrong, and the ciphertext is rejected, otherwise, no errors are expected to occur, and the decryption is expected to succeed ($d_i = 0$).

The complexity of finding the ciphertext in Step 2b is 2^{54}, and the complexity of the entire attack for 1024-bit n's is 2^{64} exponentiations on the attacker's computer. The attack requires $\log n$ chosen ciphertexts, which are decrypted on the target machine.

6 Bug Attacks on Other Schemes

6.1 Elliptic Curve Schemes

In cryptosystems based on elliptic curves, exponentiations are replaced by multiplying a point by a constant. It should be noted that the implementations of point addition (corresponding to multiplication in modular groups) and of point doubling (corresponding to squaring in modular groups) are different, but both of them use multiplications of large integers. Our bug attacks can be easily adapted in such a way that the bug is invoked only if two points are added (or alternatively, only if a point is doubled). The correctness or incorrectness of the result reveals the bits of the exponent.

6.2 Bug Attacks on Symmetric Primitives

Multiplication bugs can also be used to get information on the keys of symmetric ciphers which include multiplications, such as the block ciphers IDEA [8], MARS [5], DFC [6], MultiSwap [14], Nimbus [18] and RC6 [12], the stream cipher Rabbit [3], the message authentication code UMAC [2], etc.

In IDEA, MARS, DFC, MultiSwap and Nimbus, subkeys are multiplied by intermediate values. If an encryption (or decryption) result is known to be incorrect, an attacker may assume that one of the subkeys used for these multiplications is a, and the corresponding intermediate value is b. For example, by selecting a plaintext which contains b in a word that is multiplied by a subkey, the attacker can easily check if the value of that subkey is a.

In Rabbit, a 32-bit value is squared to compute a 64-bit result used to update the internal state of the cipher. In faulty implementations with word size 8 or 16 (which are likely word sizes for smart card implementations), faults in the stream can give the attacker information about the internal state. Similarly, the block cipher RC6 uses multiplications of the form $A \cdot (2A+1)$ for 32-bit values A, and thus multiplication bugs may cause errors in faulty implementations with word size 8 or 16. However, this is an unlikely scenario, since bugs in processors with small words are expected to cause frequent errors which can be easily discovered.

The MAC function UMAC uses multiplications of two words, both of which depend on the authenticated message. If an incorrect MAC is computed on a faulty processor, an attacker can gain information on the intermediate values of the computation.

Table 1. Summary of the Attacks Presented in This Paper

Scheme	Exp. Alg.	Attack	Sec.	Data	Pre-Comp. Time	Attack Time	Complexity for 32-bit Words*	Complexity for 64-bit Words*
Pohlig-Hellman	RTOL	CP/CC	5.1.1	$2\left\lceil\frac{\log p}{r}\right\rceil^{**}$	$\log p$	$\frac{\log p+2^r}{r}^{**}$	$2^6/2^{10}/2^{27}$	$2^6/2^{10}/2^{27}$
	LTOR	ACP/ ACC	4.1.1	$\log p$	-	$\log p$	$2^{10}/–/2^{10}$	$2^{10}/–/2^{10}$
	LTOR	CP/CC	4.1.3	$\frac{2\cdot 2^w\cdot w}{\log p}$	-	$\frac{2\cdot 2^w\cdot w}{\log p}$	$2^{28}/–/2^{28}$	$2^{61}/–/2^{61}$
RSA	CRT	CC	3	1	-	1	1/-/1	1/-/1
	RTOL	CC	5.2.1	$\left\lceil\frac{\log n}{r}\right\rceil^{**}$	$\frac{2^{2w}w^2}{\log^2 n}$	$\frac{\log n+2^r}{r}^{**}$	$2^5/2^{54}/2^{27}$	$2^5/2^{120}/2^{27}$
	LTOR	ACC	4.2.1	$\log n$	–	$2^w \cdot w$	$2^{10}/–/2^{37}$	$2^{10}/–/2^{70}$
	LTOR	CC	4.2.2	$\frac{4\cdot 2^w\cdot w}{\log n}$	–	$\frac{4\cdot 2^w\cdot w}{\log n}$	$2^{29}/–/2^{29}$	$2^{62}/–/2^{62}$
	LTOR	KP	4.2.3	$\frac{4\cdot 2^{2w}\cdot w^2}{\log^2 n}$	–	$\frac{4\cdot 2^w\cdot w}{\log n}$	$2^{56}/–/2^{29}$	$2^{122}/–/2^{62}$
RSA	RTOL	ACC	5.3.1	$\log n$	—	$\frac{2^{2w}\cdot w^2}{\log n}$	$2^{10}/–/2^{64}$	$2^{10}/–/2^{130}$
with	LTOR	ACC	4.3.1	$\log n$	—	$\frac{2^{2w}\cdot w^2}{\log n}$	$2^{10}/–/2^{64}$	$2^{10}/–/2^{130}$
OAEP	LTOR	CC	4.3.2	$\frac{4\cdot 2^w\cdot w}{\log n}$	$\frac{4\cdot 2^{2w}\cdot w^2}{\log^2 n}$	$\frac{4\cdot 2^w\cdot w}{\log n}$	$2^{29}/2^{56}/2^{29}$	$2^{62}/2^{122}/2^{62}$

KP – Known Plaintext.
CP – Chosen Plaintext; ACP– Adaptive Chosen Plaintext.
CC – Chosen Ciphertext; ACC– Adaptive Chosen Ciphertext.
w is the word size (in bits) of the faulty processor.
* Complexity is described in terms of data/pre-computation time/attack time.
** r is a parameter of the attack. The presented numbers are for $r = 2^5$.

7 Summary and Countermeasures

We have presented several chosen ciphertext attacks against exponentiation based public-key and secret-key cryptosystems, including Pohlig-Hellman and RSA. We show such attacks for the two most common implementations of exponentiation. We also discuss the applicability of these techniques to elliptic curve cryptosystems and symmetric ciphers. The attacks and their complexities are summarized in Table 1.

There are various countermeasures against bug attacks. Many protection techniques against fault attacks are also applicable to bug attacks, but we stress that due to the differences between the techniques, most of them have to be adapted to the new environment. As shown in Sections 4.3 and 5.3, and unlike the case of fault attacks, the mere knowledge that an error has occurred suffices to mount an attack, even if the output of decryption is not available. Therefore, if a decryption is found to be incorrect, it can be dangerous to send out an error message, and the correct result must be computed by other means.

Possible ways to compute the correct result include using a different exponentiation algorithm, or relying on the multiplicative property of the discussed schemes to blind the computations. When blinding is used, an attacker has no control over the exponentiated values, and they are not made available to her. Thus, even if faults occur during the exponentiation, no information is leaked. However, this method renders the system vulnerable to timing attacks, as the decryption of ciphertexts which trigger the bug take longer than decryptions which succeed in the first attempt. In order to protect the implementation from timing attacks, the original exponentiations must be blinded, so that no unblinded exponentiations are performed. Another alternative is to exponentiate modulo $n \cdot r$, where r is a small (e.g., 32-bit) prime, rather than modulo n, and reduce mod n only at the last step.

Acknowledgments

The authors would like to thank Orr Dunkelman for his comments. The first two authors were supported in part by the Israel MOD Research and Technology Unit.

References

1. Bellare, M., Rogaway, P.: Optimal Asymmetric Encryption – How to encrypt with RSA (Extended Abstract). In: De Santis, A. (ed.) EUROCRYPT 1994. LNCS, vol. 950, pp. 92–111. Springer, Heidelberg (1995)
2. Black, J., Halevi, S., Krawczyk, H., Krovetz, T., Rogaway, P.: UMAC: Fast and Secure Message Authentication. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 215–233. Springer, Heidelberg (1999)
3. Boesgaard, M., Vesterager, M., Pedersen, T., Christiansen, J., Scavenius, O.: Rabbit: A New High Performance Stream Cipher. In: Johansson, T. (ed.) FSE 2003. LNCS, vol. 2887, pp. 307–329. Springer, Heidelberg (2003)

4. Boneh, D., DeMillo, R.A., Lipton, R.J.: On The Importance of Checking Cryptographic Protocols for Faults. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 37–51. Springer, Heidelberg (1997)
5. Burwick, C., Coppersmith, D., D'Avignon, E., Gennaro, R., Halevi, S., Jutla, C., Matyas Jr., S.M., O'Connor, L., Peyravian, M., Safford, D., Zunic, N.: MARS: A Candidate Cipher for AES. In: AES—The First Advanced Encryption Standard Candidate Conference, Conference Proceedings (1998)
6. Gilbert, H., Girault, M., Hoogvorst, P., Noilhan, F., Pornin, T., Poupard, G., Stern, J., Vaudenay, S.: Decorrelated Fast Cipher: An AES Candidate. In: AES—The First Advanced Encryption Standard Candidate Conference, Conference Proceedings (1998)
7. King, S.T., Tucek, J., Cozzie, A., Grier, C., Jiang, W., Zhou, Y.: Designing and Implementing Malicious Hardware, presented in LEET 08, `http://www.usenix.org/events/leet08/tech/full_papers/king/king.pdf`
8. Lai, X., Massey, J.L., Murphy, S.: Markov Ciphers and Differential Cryptanalysis. In: Davies, D.W. (ed.) EUROCRYPT 1991. LNCS, vol. 547, pp. 17–38. Springer, Heidelberg (1991)
9. Markoff, J.: F.B.I. Says the Military Had Bogus Computer Gear, New York Times (May 9, 2008), `http://www.nytimes.com/2008/05/09/technology/09cisco.html`
10. Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press, Boca Raton (1996)
11. Pohlig, S.C., Hellman, M.E.: An Improved Algorithm for Computing Logarithms Over GF(p) and Its Cryptographic Significance. IEEE Transactions on Information Theory 24(1), 106–111 (1978)
12. Rivest, R.L., Robshaw, M.J.B., Sidney, R., Yin, Y.L.: The RC6 Block Cipher. In: AES—The First Advanced Encryption Standard Candidate Conference, Conference Proceedings (1998)
13. Rivest, R.L., Shamir, A., Adleman, L.: A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Communications of the ACM 21(2), 120–126 (1978)
14. Screamer, B.: Microsoft's Digital Rights Management Scheme – Technical Details (October 2001), `http://cryptome.org/ms-drm.htm`
15. Shamir, A., Rivest, R.L., Adleman, L.M.: Mental Poker. In: Klarner, D.A. (ed.) The Mathematical Gardner, pp. 37–43. Wadsworth (1981)
16. Shoup, V.: OAEP Reconsidered (Extended Abstract). In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 239–259. Springer, Heidelberg (2001)
17. U.S.D. of Defense, Defense science board task force on high performance microchip supply (February 2005), `http://www.acq.osd.mil/dsb/reports/2005-02-HPMS_Report_Final.pdf`
18. Warner Machado, A.: The Nimbus Cipher: A Proposal for NESSIE, NESSIE Proposal (September 2000)

A Brief Descriptions of Several Cryptosystems

A.1 The Pohlig-Hellman Cryptosystem and Pohlig-Hellman-Shamir Protocol

The Pohlig-Hellman cryptosystem [11] is a symmetric cipher. Let p be a large prime number. Alice and Bob share a secret key e, $1 \leq e \leq p-2$, $\gcd(e, p-1) = 1$.

When Alice wants to encrypt a message m, she computes $c = m^e \bmod p$. Bob can decrypt c by computing its e-th root modulo p. In practice, the decryption is performed by computing $c^d \bmod p$, where d is a decryption exponent such that $d \cdot e \equiv 1 \pmod{p-1}$. Note that given the encryption exponent e, the decryption exponent d can be easily computed, and thus e must be kept secret.

The Pohlig-Hellman-Shamir [15] keyless protocol allows encrypted communication between two parties that do not have shared secret keys. The protocol is based on the commutative properties of the Pohlig-Hellman cipher. Let p be a large prime number. Alice and Bob each has a secret encryption exponent (e_A and e_B, respectively) and a secret decryption exponent (d_A and d_B, respectively) such that $e_A \cdot d_A \equiv e_B \cdot d_B \equiv 1 \pmod{p-1}$. When Alice wishes to send Bob an encrypted message m, she sends $c_1 = m^{e_A} \bmod p$. Bob then computes $c_2 = c_1^{e_B} \bmod p$ and sends it back to Alice. Alice decrypts c_2 and sends the decryption $c_3 = c_2^{d_A} \bmod p$ to Bob. Finally, Bob decrypts c_3 to get the message $m = c_3^{d_B} \bmod p$. The protocol is secure under standard computational assumptions (The Diffie-Hellman assumption), but not against man in the middle attacks.

A.2 The RSA Cryptosystem

RSA [13] is a public-key cryptosystem. Let $n = pq$ be a product of two large prime integers. Bob has a public key (n, e) such that $\gcd(e, (p-1)(q-1)) = 1$, and a private key (n, d) such that $d \cdot e \equiv 1 \pmod{(p-1)(q-1)}$. When Alice wants to send bob an encrypted message m she computes $c = m^e \bmod n$. When Bob wants to decrypt the ciphertext he computes $c^d \equiv m^{de} \equiv m \pmod{n}$.

The security of RSA relies on the hardness of factoring n. If the factors of n are known, RSA can be easily broken.

A.3 RSA Decryption Using CRT

The modular exponentiations required by RSA are computationally expensive. Some implementations of RSA perform the decryption modulo p and q separately, and then use the Chinese remainder theorem (CRT) to compute the decryption $c^d \bmod n$. Such an implementation speeds up the decryption by a factor of 4 compared to naive implementations.

Given a ciphertext c, it is first reduced modulo p and modulo q. The two values are exponentiated modulo p and q separately: $m_p = c^{d_p} \bmod p$, and $m_q = c^{d_q} \bmod q$, where $d_p = d \bmod p-1$ and $d_q = d \bmod q-1$. Now m is computed using CRT, such that $m \equiv m_p \pmod{p}$ and $m \equiv m_q \pmod{q}$. This is done by computing $m = (xm_p + ym_q) \bmod n$, where x and y are pre-computed integers that satisfy:

$$\begin{cases} x \equiv 1 \pmod{p} \\ x \equiv 0 \pmod{q} \end{cases} \quad \text{and} \quad \begin{cases} y \equiv 0 \pmod{p} \\ y \equiv 1 \pmod{q} \end{cases} .$$

A.4 OAEP

Optimal Asymmetric Encryption Padding (OAEP) [1] and OAEP+ [16] are methods of encoding a plaintext before its encryption, with three major goals: adding randomization to deterministic encryption schemes (e.g., RSA), preventing the ciphertext from leaking information about the plaintexts, and preventing chosen ciphertext attacks. OAEP is based on two one-way functions G and H, which are used to create a two-round Feistel network, while OAEP+ uses three one-way functions. Only OAEP is described here.

Let $G : \{0,1\}^{k_0} \rightarrow \{0,1\}^{l+k_1}$, $H : \{0,1\}^{l+k_1} \rightarrow \{0,1\}^{k_0}$ be two one-way functions, where l is the length of the plaintext, and k_0, k_1 are security parameters. When Alice wants to compute the encryption C of a plaintext M, she chooses a random value $r \in \{0,1\}^{k_0}$ and computes

$$\begin{aligned} s &= G(r) \oplus (M||0^{k_1}), \\ t &= (H(s) \oplus r), \\ w &= s||t, \\ C &= E(w), \end{aligned}$$

where $||$ denotes concatenation of binary vectors, and E denotes encryption with the underlying cipher. Decryption of c is performed by:

$$\begin{aligned} w &= D(C), \\ s &= w[0 \ldots l + k_1 - 1], \\ t &= w[l + k_1 \ldots n - 1], \\ r &= H(s) \oplus t, \\ y &= G(r) \oplus s, \\ M &= y[0 \ldots l - 1], \\ z &= y[l \ldots l + k_1 - 1], \end{aligned}$$

where D denotes decryption under the same cipher used in the encryption phase. If $z \neq 0^{k_0}$, then the ciphertext is rejected and no plaintext is provided. Otherwise, the decrypted plaintext is M.