J Cryptol (2016) 29:775–805
DOI: 10.1007/s00145-015-9209-1

Journal of
CRYPTOLOGY

Bug Attacks

Eli Biham · Yaniv Carmeli
Computer Science Department, Technion - Israel Institute of Technology, Haifa 3200003, Israel
biham@cs.technion.ac.il; yanivca@cs.technion.ac.il
http://www.cs.technion.ac.il/~biham/; http://www.cs.technion.ac.il/~yanivca/

Adi Shamir
Computer Science Department, The Weizmann Institute of Science, Rehovot 7610001, Israel
adi.shamir@weizmann.ac.il

Communicated by Hugo Krawczyk.

Received 13 April 2012
Online publication 7 August 2015

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 and ElGamal encryption 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. As shown by recent revelation of top secret NSA documents by Edward Snowden, intentional hardware modifications is a method that was used by the USA to weaken the security of commercial equipment sent to targeted organizations.

Keywords. Bug attack, Fault attack, RSA, Pohlig–Hellman, ElGamal encryption.

1. Introduction

With the increasing word size and the sophisticated optimizations of multiplication units in modern microprocessors, it becomes increasingly likely that they contain 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 allows memory corruptions outside the permitted

range of writing for a process), etc. A non-exhaustive list of known hardware bugs is given in Appendix 2.

In this paper we show that a bug in the microprocessor that is used to carry out cryptographic computations can be exploited by an attacker to learn secret information about the cryptographic keys. 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 computers 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. Similar attacks are probably 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 might be 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 et al. [8] 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 outside its operating envelope (temperature, voltage, clock frequency, etc.), or subjecting it to clock and voltage glitches or light and laser pulses. 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 halfway around the world, 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 for 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, and 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 PCs 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 phones are running RSA or elliptic curve computations on signal processors made by TI and others. FPGA or ASIC devices embed 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 the correctness.

In addition to such innocent bugs, there is the issue of intentionally tampered hardware, which is a major security problem. Even commercially sold bug-free processors can be made buggy by anyone along the supply chain who modifies them. In February 2005, the matter was addressed in a US Department of Defense (DoD) report [30], which warned about the risks of importing hardware from foreign countries to the USA. NSA documents leaked by Edward Snowden provide evidence that the NSA does exactly that. In December 2013, Der Spiegel ran a story [29] based on the leaked documents in which they reveal the method referred by the NSA as *interdiction*: When an NSA target orders new electronic equipment, the NSA diverts the shipments to pass through their own workshops, where the packages are opened and the equipment is tampered with. Other leaked documents [2] provide insight into some of the ways used by the NSA in order to tamper with the equipment, including adding their own software, replacing the existing firmware and even completely replacing the hardware.

In this paper, we show 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.

We present bug attacks against the widely deployed RSA [24] cipher, against the Pohlig–Hellman cipher [22], the ElGamal [12] encryption, and against several implementations of those ciphers.[1] We show that the secret exponents 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) [19, Note14.70], the public modulus n can be factored using a single chosen ciphertext. A particularly interesting observation is that even though RSA-OAEP [4] 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 computing the result twice with a different exponentiation algorithm to verify the result may protect the scheme against fault attacks, but will leak the full key via a bug attack. We also discuss the possible applicability of this technique to elliptic curve schemes and several symmetric primitives.

This paper is organized as follows: Sect. 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 CRT, using a single chosen ciphertext. Section 4 presents attacks on several cryptosystems when exponentiations are computed using the LTOR algorithm, and Sect. 5 presents attacks on the same schemes when the exponentiations are com-

[1] We present a reduction that converts our attacks on the Pohlig–Hellman cipher to attacks on ElGamal encryption.

puted using the RTOL algorithm. Section 6 describes attacks which use the Legendre symbol to identify erroneous Pohlig–Hellman decryptions. In Sect. 7, we discuss the applicability of bug attacks to elliptic curve schemes and some symmetric primitives. Section 8 summarizes the contributions of this paper and presents the time and data complexities of all our attacks. In Appendix 1, we provide descriptions of the cryptosystems discussed in this paper. Finally, Appendix 2 includes a non-exhaustive list of known hardware bugs.

An extended abstract of this paper was published in the proceedings of CRYPTO 2008 [5]. In particular, we added attacks based on the Legendre symbol, referred explicitly to ElGamal encryption, and updated and revised the content and presentation in the rest of the paper.

2. Overview of Our Methods and Notations

We present several attacks which use multiplication bugs. We concentrate on multiplication since, on one hand, it is a common operation in cryptographic computations, and on the other hand it is typically a complex operation and its implementations are aggressively optimized. Therefore, bugs are much more likely to exist in multiplication instructions than in simple operations like addition or XOR and are less likely to be discovered by the manufacturers. Furthermore, these complex operations are more likely to fail when the processor operates under unusual circumstances, such as overclocking, even when no bugs exist under normal operating conditions.

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 $\mathbf{x}\cdot\mathbf{y}$ to denote the result of multiplying x by y on a bug-free processor and $\mathbf{x}\odot\mathbf{y}$ to denote the result of the same computation when performed on a faulty processor. Similarly, the notation $\mathbf{x^l}$ denotes the value of x to the power l as computed on a bug-free processor, and $\mathbf{x^{\langle l \rangle}}$ denotes the value of x to the power l as computed by a particular algorithm on a faulty processor (see Sect. 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. When no errors occur, 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, as determined by the bits of the secret exponent d. The results of those decryptions are used to retrieve the bits of 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 some 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. But if x is first multiplied by a different value y (as controlled by d), 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 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 during the decryption process (due to the bits of d), we expect the decryption result to be correct.

2.4. *Complexity Analysis*

Let w be the length (in bits) of the words of the processor. In the analysis of the complexity 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 about 2^{-27}, and for $w = 64$ it is about 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}.

It is important to note that obtaining the data required for some of our attacks might be slow, as in some cases, the attack requires a vast number of decryptions on the attacked faulty processor.

2.5. *Exponentiation Algorithms*

Given a value x and a secret exponent $d = d_{\log n} d_{\log n - 1} \ldots d_1 d_0$ (where $d_i \in \{0, 1\}$ are the binary digits of d, i.e., $d = \sum_{i=0}^{\lfloor \log n \rfloor} d_i 2^i$), the exponentiation $x \mapsto x^d \mod n$ can be efficiently computed by several exponentiation algorithms [19, Chapter 14.6]. In this paper, we present attacks against implementations that use the two basic exponentiation algorithms, LTOR and RTOL, described in Algorithm 1. Several of our techniques can

LTOR Exponentiation

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

RTOL Exponentiation

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

Algorithm 1: The Two Basic Exponentiation Algorithms

be easily adapted to attack implementations that use other exponentiation algorithms such as the sliding window algorithm and the k-ary exponentiation algorithm.

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 may be 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 when she devises the attack.
2. Given a value n, the number of bits in the binary representation of n is $\lfloor \log_2 n \rfloor + 1$ (the indices of the bits of n are $0, \ldots, \lfloor \log n \rfloor$, where 0 is the index of the least significant bit, and $\lfloor \log n \rfloor$ is the index of the most significant bit). Throughout this paper, we use $\log n$ (without the floor operator) as a shorthand for the index of the most significant bit of n.
3. It may be the case that more than one pair of buggy inputs a and b exist. In such cases, if $\gamma > 1$ 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 about $\min(\gamma, \lfloor (\log n)/w \rfloor)$ (It is enough that one of the pairs of the buggy inputs appears in order to cause an error in the computation, it does not matter which of them. Still, we cannot use more input pairs than the number of words in the representation of the big integer). For example, if two pairs of buggy inputs are known, $a - b$ and $c - d$, we can choose the operand we can control to contain both a and c, and then we need either b or d to appear randomly in the other operand. If some of the buggy pairs of operands share the same value for one of the operands, this factor can even be 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 for this remark 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 the cases of (x^2 or xy) relies on whether $a \neq b$.

3. Breaking CRT-RSA with One Chosen Ciphertext

We now describe a simple attack on RSA implementations in which decryptions are performed using the 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, $C_p = C \mod p$ is randomized modulo the smaller factor p, but $C_q = C \mod q = C$ 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 in the original fault attack of [8], 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 ($p \leq C < q$) 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 (rather than using CRT).

We first present chosen plaintext (or chosen ciphertext) attacks against the Pohlig–Hellman scheme and ElGamal encryption, 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 and ElGamal Encryption*

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.

The same attacks can also be applied to retrieve the secret exponent x of the ElGamal encryption system by the following reduction:

- Each call to a Pohlig–Hellman decryption of a ciphertext C is replaced by a decryption of (C, r) for a randomly chosen r.
- Denote the result of the ElGamal decryption by Y, and then the Pohlig–Hellman decryption of C is replaced by $M = Y^{-1} \cdot r \mod p$.

The same reduction applies also for computations on a faulty processor, where the decryption of ElGamal should be performed on the same faulty processor.

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 $\log p + 1$ 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. Algorithm 2 describes the attack.

The attack is based on the following observations. Since p is a known prime, the attacker can compute arbitrary roots modulo p. 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 a modular square root of X. At the beginning of the next iteration z is squared (and thus its value becomes 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 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 3d may be computed on the buggy microprocessor by using the multiplicative property of modular exponentiation. The attacker may request the decryption M' of $C' = C^3 \mod p$ (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 whether an error had occurred. Thus, no calls to a bug-free decryption device that uses the same secret key (which is usually unavailable) are required. In fact, since the same value of X is used for each of the iterations, the correct

1. Choose a value X which contains the words a and b.
2. Set $d_{\log p} = 1$
3. For $i = \log p - 1$ down to 1 do
   # Trying to recover bit d_i, given that bits $d_{i+1} \ldots d_{\log p}$ are known.
   (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 $C = X^{1/2d'} \mod p$.
   (c) Ask for the decryption $\hat{M} = C^{\langle d \rangle} \mod p$ on the faulty processor.
   (d) Obtain the correct decryption $M = C^d \mod p$ (we later describe how this step can be performed even without access to a bug-free processor).
   (e) If $M = \hat{M}$ conclude that $d_i = 1$, otherwise conclude that $d_i = 0$.
4. Set $d_0 = 1$.

Algorithm 2: Basic Adaptive Chosen Ciphertext Attack Against Pohlig–Hellman with LTOR

decryption M can be computed from the value of the correct decryption in the previous iteration as: $M = \bar{M}^{d'/\bar{d}'} \mod n$, 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 (we later describe how to find such X). Using this observation, we reduce the expected complexity of retrieving d by a constant factor. A further improvement uses values of X which contain a and b, such that when X is squared m times repeatedly on a faulty processor (for some $m > 0$), all the intermediate squares $X^{\langle 2^j \rangle}$ (for $0 \leq j \leq m$) contain a and b. The faulty squares of X as computed by the faulty processor are $X^{\langle 2 \rangle}, X^{\langle 2^2 \rangle}, \ldots, X^{\langle 2^m \rangle}$. Let $\beta_j = X^{2^j} / X^{\langle 2^j \rangle}$ for $0 \leq j \leq m$. Using such values for X and assuming uniform and independent distribution of the bits of d, we can improve the expected complexity of the attack by a factor of $\alpha = 2 - 2^{-m}$. The trick is to identify the length of a subsequence of several consecutive zero bits of d using one chosen ciphertext.

In this improved attack, the bits of d are retrieved starting from the most significant bit, as in the original attack. The attack is described in Algorithm 3.

As in the basic attack, the correct decryption of C can be obtained by a blinded decryption query to the faulty processor, which is not likely to trigger the fault.

The attack chooses ciphertexts such that after $\log p - i$ iterations of the exponentiation (i.e., decryption), the intermediate value of z in the LTOR algorithm is a square root of X. At the beginning of the next iteration, its values are squared (and thus it is now X). Then, if $d_i = 1$, X is multiplied by C and therefore no errors are expected to occur during the algorithm (and we get $j = 0$ in Step 3f of the attack). Otherwise, $d_i = 0$, X is

1. Choose X such that $X, X^{\langle 2\rangle}, X^{\langle 2^2\rangle}, \ldots, X^{\langle 2^m\rangle}$ all contain a and b (see below how to implement this step in the most efficient way).
2. Set $d_{\log p} = 1$
3. While not all the bits $d_{\log p-1}, \ldots, d_2, d_1$ are known:
   (a) Let i be the smallest index such that all the bits $d_{i+1}, \ldots, d_{\log p}$ are already known.
   (b) Denote the value of the known bits of d by $d' = \sum_{k=i+1}^{\log p} 2^{k-(i+1)} d_k$.
   (c) Compute $C = X^{1/(2d')} \mod p$.
   (d) Ask for the decryption $\hat{M} = C^{\langle d\rangle} \mod p$ on the faulty processor.
   (e) Obtain the correct decryption $M = C^d \mod p$.
   (f) Find j such that $M/\hat{M} = \beta_j{}^{2^{i-j}} \mod p$ ($j \in \{0, \ldots, m\}$).
   (g) Conclude that the next j bits of d are zero (i.e., $d_i = d_{i-1} = \ldots = d_{i-(j-1)} = 0$), and if $j < m$, the following bit is one (i.e., $d_{i-j} = 1$).
4. Set $d_0 = 1$.

Algorithm 3: Improved Adaptive Chosen Ciphertext Attack Against Pohlig–Hellman with LTOR

squared, and the result is $X^{\langle 2\rangle}$ (due to the bug). If $d_{i-1} = 1$, then the intermediate result is multiplied by C, and therefore no additional computation errors are expected to occur. In such a case, the output of the exponentiation algorithm is $\hat{M} = (X^{\langle 2\rangle})^{2^i} C^*$, where C^* is a value which depends only on C and on the unknown bits of d. For similar reasons, the correct exponentiation of C is $M = (X^2)^{2^i} C^*$, so we get $M/\hat{M} = \beta_1{}^{2^i}$. Moreover, if there are j ($j \leq m$) consecutive zero bits, they are successfully identified by the condition in Step 3f, as $M = (X^2)^{2^{i-j}} C^*$ and $\hat{M} = (X^{\langle 2\rangle})^{2^{i-j}} C^*$ for some appropriate C^*, and we get $M/\hat{M} = \beta_1{}^{2^{i-j}}$. Note that the length of sequences of zeros we can identify in this way is bounded by m, and that if we identify a sequence of m zero bits, we cannot determine whether the following bit (after the sequence) is set or not.

Each iteration of the attack retrieves at least one bit of d and may retrieve up to m bits of d. Assuming a uniform independent distribution of the bits of d, the expected number of bits retrieved in each iteration is $\alpha = \sum_{k=0}^{m} 2^{-k} = 2 - 2^{-m}$, which for $m \geq 1$, is in the range $\alpha \in [1.5, 2)$. Therefore, on average $\log p/(2 - 2^{-m}) + 1$ chosen ciphertexts are required for the attack, which is 1.5–2 times more efficient than the basic version.

Finding X. For a general m, finding such values of X as required for the attack may be hard, because the probability that the square of a random value which contains a and b also contains a and b is very low. However, for $m = 2$ and $m = 3$, when $w = 32$, we successfully found such values: When $p > 2^{256}$ we can use $X = 2^{127} + 2^{32}a + b$, for which both X and $X^{\langle 2\rangle}$ contain a and b. For $p > 2^{893}$, we can use $X = 2^{223} + 2^{96}(2^{32}a + b) + 2^{10}$, for which all three values X, $X^{\langle 2\rangle}$ and $X^{\langle 4\rangle}$ contain both a and b.

By investing a computation time of about 2^{54} in each iteration of the attack, we can reduce the data complexity by a factor of 2. We search for values X such that $X, X^{\langle 2\rangle}$, and $(X \odot X^{1/2d'})^{\langle 2\rangle} = (X \odot C)^{\langle 2\rangle}$, all contain a and b (we can choose values X that satisfy the first two conditions, while the appearance of a and b in $(X \odot C)^{\langle 2\rangle}$ has a probability of about 2^{-54} to occur randomly). Using such values, we can retrieve two bits of d in

every iteration of the attack, reducing the number of calls to the faulty decryption oracle by a factor of 2. However, the total time complexity of the attack increases to about 2^{63}.

4.1.3. *Chosen Ciphertext Attack*

The (non-adaptive) chosen ciphertext attack presented later in Sect. 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*

In this section, 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 $C^d \mod n$ 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.

In addition, unlike the case of Pohlig–Hellman, in RSA it is easy to check whether the decrypted message $\hat{M}$ is the correct decryption of a chosen ciphertext C by checking whether $\hat{M}^e \equiv C \pmod n$. Therefore, 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}. The details are provided in Algorithm 4.

The attack is similar to the basic attack presented in Sect. 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: Now an incorrect result corresponds to $d_i = 1$ and a correct result corresponds to $d_i = 0$.

During the execution of the LTOR algorithm, the intermediate value of the variable z during iteration $\log n - i$ contains a (due to the selection of C in Step 2b 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 2e 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 2e.

As explained in Sect. 2, the probability that the random number $C^{d'} \mod n$ contains somewhere along it the word a is 2^{-27} (for our standard parameters). Therefore, Step 2b takes an average time of 2^{27} exponentiations on the attacker's computer.

1. Set $d_{\log n} = 1$
2. For $i = \log n - 1$ 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 $C^{2d'} \mod n$ contains a.
   (c) Ask for the decryption $\hat{M} = C^{\langle d \rangle} \mod n$ using the faulty processor.
   (d) Compute $\hat{C} = \hat{M}^e \mod n$.
   (e) If $\hat{C} = C$ conclude that $d_i = 0$, otherwise conclude that $d_i = 1$.
3. Set $d_0 = 1$.

Algorithm 4: Adaptive Chosen Ciphertext Attack Against RSA with LTOR

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. Set $d_{\log n} = 1$
3. For $i = \log n - 1$ 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 $X_j = C_j^{2d'} \mod n$.
   (c) Consider all ciphertexts C_j such that X_j contains a:
      i. If for all such ciphertexts C_j it holds that $\hat{M}_j^e \mod n = C_j$ then set $d_i = 0$.
      ii. If for all such ciphertexts C_j it holds that $\hat{M}_j^e \mod n \neq C_j$ 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, $\hat{M}_j^e \mod n = C_j$ and for others $\hat{M}_j^e \mod n \neq C_j$ (i.e., a previously set value of one of the bits is wrong) then backtrack.
4. Set $d_0 = 1$.

Algorithm 5: Chosen Ciphertext Attack Against RSA with LTOR

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 (of the 2^{29} for which the intermediate value of z in round i 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 Sect. 4.2.1. If for some i there are no ciphertexts C_j for which $X_j = C_j^{d'} \mod n$ 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. The attack is presented in Algorithm 5.

We remark that there is a trade-off between the number of ciphertexts and the time complexity of the attack. With more data, there is a higher probability that there will be some ciphertext C_j for which X_j contains a, for some iteration i of the attack, and the time complexity decreases. On the other hand, with less data this probability is lower, and when there are no such ciphertexts, we have to continue the attack both with $d_i = 0$ and $d_i = 1$ [Step 3(c)iii], thus increasing the attack time. 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 Sect. 4.2.2 can be easily transformed into a known plaintext attack which requires 2^{56} known plaintexts. Among the 2^{56} plaintexts, only $2^{56} \cdot 2^{-27} = 2^{29}$ 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 of Algorithm 5 (Sect. 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 protected by modes of operation. The most popular mode is OAEP [4], which provides provable security. 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 (e.g., contains the word a, or such that some intermediate value of the decryption contains a or b), 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 Sect. 4.2, with some minor modifications. The same attacks are also applicable to OAEP+ [28].

4.3.1. *Adaptive Chosen Ciphertext Attack*

Unlike the attack of Algorithm 4 (Sect. 4.2.1), OAEP stops us from directly choosing ciphertexts C which contain b, and thus in Step 2b, we must choose random messages (on our own computer) until b "appears" in C at random. As explained in Sect. 2.4, the probability that this happens and also $C^{2d'} \mod n$ contains a is 2^{-54}. As mentioned above, computation errors are identified in Step 2e 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 Sect. 4.2.2 (Algorithm 5) 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} precomputation 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 and ElGamal encryption, 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 and ElGamal Encryption*

We present a chosen ciphertext attack against Pohlig–Hellman implementation in which exponentiations are performed using RTOL. The attack is aimed at retrieving the bits of the secret exponent d. As in Sect. 4.1, an identical chosen plaintext attack can retrieve the bits of the secret exponent e. Also, the attacks on Pohlig–Hellman described here can be applied to the ElGamal encryption system by the same reduction described in Sect. 4.1.

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

Algorithm 6: Chosen Ciphertext Attack Against Pohlig–Hellman with RTOL

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 Sect. 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 presented in Algorithm 6.

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 only if $d_{i+1} = 1$. 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 Sect. 4.1, all the error-free decryption queries in Step 1d can be replaced by the decryption of one additional ciphertext on the faulty processor: The attacker can request the decryption M^3 of $C^3 \mod p$ (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

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

1. Choose a random ciphertext C_0, and let $t = 0$.
2. While $t \leq \log n$ and C_t does not contain both a and b do:
   (a) $t = t + 1$.
   (b) Compute $C_t = C_{t-1}^2 \mod n$.
3. Let $X = C_t$ and let $X^{\langle 2 \rangle}$ be the result of squaring X on a faulty processor.
4. Let $\beta = X^2 / X^{\langle 2 \rangle} \mod n$.
5. For $i = \log n - (\log n \mod r)$ down to 0 step $-r$
   (a) Ask for the decryption $\hat{M}$ of $C = C_{t-i}$ using the faulty processor, $M = C_{t-i}^{\langle d \rangle} \mod p$.
   (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 $\hat{C} = \hat{M}^e \mod n$.
   (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 $d_{i+k} = u_k, \ \forall\, 0 \leq k < r$.

Algorithm 7: Chosen Ciphertext attack against RSA with RTOL

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 $r \geq 12$, however, this computation takes longer, and there is a trade-off 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 (Sect. 5.1.1, Algorithm 6), 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 Algorithm 7). The parameter r represents the number of bits retrieved in each iteration. Algorithm 7 describes the attack.

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,

1. Set $d_0 = 1$.
2. For $i = 1$ to $\log n$
   (a) Denote the value of the known bits of d by $d' = \sum_{k=0}^{i-1} 2^k d_k \mod n$.
   (b) Repeatedly encrypt random messages M until $C = E(M) = (\text{OAEP}(M))^e$ satisfies that $C^{2^i} \mod n$ contains a and $C^{d'} \mod n$ 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 $d_i = 1$.

Algorithm 8: Adaptive Chosen Ciphertext Attack Against RSA-OAEP with RTOL

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 Sect. 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 Sect. 5.1.1) for large values of r, there is a trade-off 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 on RSA-OAEP from Sect. 4.3, but it identifies the bits of d starting from the *least* significant bit. The details are provided in Algorithm 8.

After i iterations of the decryption exponentiation algorithm, the value of the variable z is $C^{d'} \mod n$, and the value of the variable y is $C^{2^i} \mod n$. 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 Using the Legendre Symbol and Square Roots

In this section, we describe techniques which use the Legendre symbol to identify incorrect decryptions (in the case of exponentiations with RTOL), and help identify the bits of the secret exponent (in the case of exponentiations with LTOR). Using these techniques,

we are able to mount known plaintext bug attacks on the Pohlig–Hellman scheme, which were not possible with the techniques of Sects. 4 and 5.

As in Sects. 4.1 and 5.1, the attacks on Pohlig–Hellman presented here can be converted to attacks on ElGamal encryption. Because ElGamal is a public-key encryption scheme, we have to address explicitly the case of known plaintext attacks: these attacks are converted into attacks on ElGamal encryption in which the ciphertexts are randomly selected without control of the attacker, and both the ciphertexts and the decrypted plaintexts become known to the attacker.

6.1. *Bug Attacks on Pohlig–Hellman Implementations that Use RTOL*

Exponentiation by an odd exponent modulo a prime p preserves the Legendre symbol of the input. In the case of Pohlig–Hellman, since the decryption exponent d is odd, for every ciphertext C:

$$\left(\frac{C}{p}\right) = \left(\frac{C^d}{p}\right) .$$

We observe that if a bug occurs when exponentiating with RTOL, the Legendre symbol of the (faulty) decrypted message may be different than the Legendre symbol of the ciphertext, and thus the Legendre symbol can be used to ascertain that a bug had occurred. Consider the exponentiation of a ciphertext C with RTOL: the least significant bit of d is one, thus after the first iteration of the exponentiation $z = C$, and $y = C^2 \mod p$. From now on the Legendre symbol of y modulo p is always one (y is the result of a square operation, and thus is a quadratic residue), and since z can only be multiplied by y, the Legendre symbol of z modulo p does not change in the remaining iterations. However, if as a result of the execution of the buggy instruction the result of squaring y has a Legendre symbol -1, then the Legendre symbol of the decrypted message will be flipped.

We first present a chosen ciphertext (or chosen plaintext) attack against Pohlig–Hellman where exponentiations are performed with RTOL, and then extend it to a known plaintext attack.

6.1.1. *Chosen Ciphertext Attack*

The chosen ciphertext attack presented in this section uses the technique described above to find the bits of the secret exponent d. As in the attacks on Pohlig–Hellman from Sects. 4.1 and 5.1, a similar chosen plaintext attack can retrieve the bits of the secret exponent e. In the following attack, the bits of d are retrieved from the least significant bit to the most significant (note that in principle the bits of d may be retrieved in any order by this attack, but this order may be implemented more efficiently than others with fewer square root calls).

The attack uses a value X which contains both a and b, such that the Legendre symbol of $X^{\langle 2 \rangle} \mod p$ is -1. Such a value can be easily obtained by selecting random numbers which contain a and b and checking the Legendre symbol of $X^{\langle 2 \rangle} \mod p$. Since half the

1. Choose X such that X contains a and b, and $\left(\frac{X^{\langle 2\rangle}}{p}\right) = -1$.
2. Set $d_0 = 1$.
3. For $i = 1$ to $\log p$
   (a) Compute $C = X^{1/2^i} \mod p$ (any 2^i'th root is accepted).
   (b) Ask for the decryption $\hat{M} = C^{\langle d\rangle} \mod p$ on the faulty processor.
   (c) If $\left(\frac{C}{p}\right) = \left(\frac{\hat{M}}{p}\right)$ set $d_i = 0$, otherwise set $d_i = 1$.

Algorithm 9: Chosen Ciphertext Attack Against Pohlig–Hellman with RTOL

numbers in Z_p^* have a Legendre symbol of -1, a suitable value is expected to be found after two attempts on average. The attack is presented in Algorithm 9.
The attack is as follows: The attack requires decryption of $\log p$ ciphertexts, $\log p$ extractions of modular roots and $\log p$ computations of Legendre symbol.

6.1.2. *Known Plaintext Attack*

Using the Legendre symbol to identify incorrect decryptions an attacker can retrieve the bits of the secret exponent d (if the plaintexts were obtained by decrypting the ciphertexts on a faulty processor) or the secret exponent e (in case the ciphertexts are the result of encrypting the plaintexts on the faulty processor). Without loss of generality we describe our attack against the former case.

The attack requires 2^{57} plaintexts and ciphertexts. For every $0 \leq i \leq \log p$ we expect that for about eight ciphertexts, the value C^{2^i} contains both a and b, and that about half of them also have a Legendre symbol -1 when squared modulo p on the buggy processor. These ciphertexts can be used in an attack similar to the one described in Algorithm 9 (Sect. 6.1.1).

The bits of d can be retrieved in any order, thus if there are no suitable ciphertexts to retrieve a specific bit, an attacker can continue to retrieve the rest of the bits, and guess the value of the missing bit at the end. More than 98 % $(1 - e^{-4})$ of the bits are expected to be successfully retrieved by this method with this number of ciphertexts (e.g., for 512-bit moduli only about 10 remain to be tried at the end). The detailed attack is described in Algorithm 10.

In addition to the previous method, another method can be used to retrieve the bits of d, using the same data. Unlike the first method, this second method requires retrieving the bits of d in a specific order, from the least significant to the most significant. We expect that among the 2^{57} ciphertexts, for every $0 \leq i \leq \log p$ there are about eight ciphertexts such that after i iterations of exponentiation the value of the variable z (of RTOL algorithm) contains a, and the value of the variable y contains b (as in the attack from Sect. 5.3.1). We expect that about half of them satisfy $\left(\frac{z \odot y}{p}\right) \neq \left(\frac{C}{p}\right)$, and thus also $\left(\frac{\hat{M}}{p}\right) \neq \left(\frac{C}{p}\right)$. When $d_0, d_1, \ldots, d_{i-1}$ are known, these ciphertexts can be easily identified. If at least one such ciphertext exists, the bit d_i can also be identified by this observation. Both described methods can be applied together using the same data in order to reduce the probability of an error (or slightly reduce the data complexity).

1. Set $d_0 = 1$.
2. For each $\hat{M}, C$ of the 2^{57} message-ciphertext pairs
   (a) For $i = 1$ to $\log p$
      i. If C^{2^i} contains a and b, and $\left(\frac{C^{2^{i+1}}}{p}\right) = -1$
         A. If $\left(\frac{C}{p}\right) = \left(\frac{\hat{M}}{p}\right)$ set $d_i = 0$.
         B. Otherwise set $d_i = 1$.
3. Exhaustively search for the values of all the bits of d that were not found in Step 2.

Algorithm 10: Known Plaintext Attack Against Pohlig–Hellman with RTOL

6.2. *Bug Attacks on Pohlig–Hellman Implementations that Use LTOR*

In this section, we describe techniques that use extraction of modular square roots in order to identify the bits of the secret exponent of Pohlig–Hellman. We open this section with a general discussion of long multiplications in the presence of a multiplication bug.

Let p be a large prime, and let $C \in Z_p^*$ be some number which contains b. Consider the functions $f, \hat{f} : Z_p^* \to Z_p^*$ defined by $f(X) = X \cdot C \mod p$ and $\hat{f}(X) = X \odot C \mod p$. While f is an automorphism, $\hat{f}$ is not. Due to the buggy multiplication there are some values X, X^* such that $\hat{f}(X) = \hat{f}(X^*)$ (for example, since X^* contains a and X does not). As a result, there are some values in Z_p^* which cannot be the result of a buggy multiplication by C. We show that given a number V, it is possible to estimate how many pre-images $\hat{f}$ has for V by inverting the buggy multiplication (assuming at most one occurrence of the bug).

Let $V = f(X) = X \cdot C$ and $\hat{V} = \hat{f}(X) = X \odot C$, for some number X which contains a, and recall that a multiplication of two big numbers is performed by multiplying every word of X by every word of C, and summing up the results with the appropriate left shifts.

Given some $\hat{V} \in Z_p^*$, it is easy to check whether $\hat{V}$ can be the result of a bug-free multiplication by C, simply by computing $X = (\hat{V} \cdot C^{-1}) \mod p$. If X does not contain a, then no bugs are expected to occur when multiplying $X \cdot C \mod p$, and $\hat{f}(X) = \hat{V}$ (and also with $f(X) = \hat{V}$). An important conclusion of this discussion is that all but about 2^{-27} of the numbers in Z_p^* can be images of $\hat{f}$ which are obtained with no executions of the bug (this value was computed using our standard parameters).

It is also possible to check whether $\hat{V}$ can be an image of $\hat{f}$ which is obtained by a multiplication with exactly one occurrence of the bug. The additive error $\delta = \hat{V} - V \mod p$ introduced into the product is a function of $s, t \in \{0, 1, \ldots, \lfloor(\log p)/w\rfloor\}$, the word locations of the words a and b in X and C, respectively, where w is the size of the word and where the least significant word is considered as location 0. The additive error as a function of s, t is

$$\delta = \delta_{s,t} = (a \odot b - a \cdot b)2^{w(s+t)} \mod p. \tag{1}$$

Furthermore, $s + t$ is limited to the range $\in \{0, 1, \ldots, 2\lfloor(\log p)/w\rfloor\}$ (there are fewer possibilities if s and/or t are known). We conclude that there are at most $2\lfloor(\log p)/w\rfloor+1$ possible values for δ. Given $\hat{V}$ and C, and assuming only one occurrence of the bug, there are $\lfloor(\log p)/w\rfloor + 1$ possible values of s (t is known because C is known). For each of the possible values of s the corresponding values of $\delta_{s,t}$, V and $X = Z \cdot C^{-1} \mod p$ can be easily computed. The correct values of s, $\delta_{s,t}$, V and X can now be easily identified, since only the correct value of X is expected to contain the word a in location s. It follows from this discussion that a fraction of 2^{-27} of the numbers in Z_p^* can be results of multiplication by C with one execution of the buggy instruction. We denote the set of those numbers by W.

Let $\beta = \hat{V}/V \mod p$ be the multiplicative error in the computation of V due to the bug. The relation between β and δ is given by:

$$\delta \equiv V\left(\frac{\hat{V}}{V} - 1\right) \equiv V\,(\beta - 1) \pmod{p}. \tag{2}$$

We now use these observations to mount chosen plaintext (or chosen ciphertext) and known plaintext attacks on Pohlig–Hellman implementations, that use the LTOR algorithm for decryption.

6.2.1. *Chosen Ciphertext Attack*

The following chosen ciphertext attack has two parts. The first part uses the techniques described above to identify some of the 1's of the secret exponent d, while the second part searches for the values of the rest of the bits. A similar chosen plaintext attack can retrieve the bits of the secret exponent e. The attack requests the decryption of 2^{24} ciphertexts which contain b.

In the first part of the attack, each incorrect decryption reveals a bit d_j of d with value $d_j = 1$. In the $(\log n - j)$-th iteration of the LTOR algorithm (the iteration that computes $z \leftarrow z^2 C^{d_j}$), if $d_j = 1$ and z^2 contains a, then we expect that $z^2 \cdot C \neq z^2 \odot C$. Let $V = z^2 \cdot C$ and $\hat{V} = z^2 \odot C$, and let $\beta = \hat{V}/V$ and $\delta = \hat{V} - V$ (all the computations are performed modulo p). The values of V, $\hat{V}$, β and δ are related by (2). The LTOR algorithm ends after j additional iterations. On a faulty processor its result is expected to be $\hat{M} = \hat{V}^{2^j} C^* \mod p$, for some C^* which depends on the value of C and on $d_{j-1}, \ldots, d_1, d_0$, while the correct result is $M = V^{2^j} C^* \mod p$ (for the same value of C^*). Let $Q \equiv \hat{M}/M \equiv \beta^{2^j} \pmod{p}$. Given any incorrect decryption $\hat{M} = C^{\langle d\rangle} \mod p$ and the corresponding correct decryption $M = C^d \mod p$ of a ciphertext C containing b in location t, this attack uses $Q \equiv \hat{M}/M \equiv \beta^{2^j} \pmod{p}$ to search for the combination of j and δ which corresponds to the computation error. For each possible combination of j and s, we first extract the 2^j-th roots of Q (there are $\gcd(2^j, p - 1)$ such roots) to determine possible values for β. For each candidate for β, we use $\delta_{s,t}$ to compute the value of V according to (2):

$$V \equiv \delta_{s,t}(\beta - 1)^{-1} \pmod{p}.$$

Once we try the correct combination of β and s, we find that $z^2 = VC^{-1} \mod p$ indeed contains a in location s. We can thus conclude that $d_j = 1$, and save the tuple (C, z^2, j) for a later stage. The probability that VC^{-1} will contain a in location s for an incorrect combination of β and s is 2^{-w}. However, the probability that this will occur during the entire course of the attack is bounded from above by $\gcd(2^{\lceil \log p \rceil}, p-1) \cdot \log^2 p/(w \cdot 2^w)$, which is usually small. For example, for $p \equiv 3 \pmod 4$ and our standard parameters, the probability of an error in the course of the attack is bounded from above by 2^{-16}.

In the second part of the attack, after identifying some of the 1's in d, we search for the values of the other bits, using the information gathered in the first part. For every $d_j = 1$ that was found in the first part of the attack, we also learnt the intermediate value of z^2 after j iterations of decrypting C_j with the LTOR algorithm. We sort the tuples $(C, ?, j)$ in descending order of j, and get intervals which start and end with bits of d with value 1 (with unknown bit values in between them). We then analyze those intervals of unknown bits in order from left to right. We recover the values of the bits in each interval by exhaustively searching for their value until the correct intermediate value of the exponentiation is received. For example, if $n = 10$ and in the first phase of the attack we identified that the value of bits d_6 and d_3 is 1, then in the second phase we first try all values of d_9, d_8, d_7 and find the correct ones, then search for the values of d_5, d_4 and finally the values of d_2, d_1, d_0.

The complexity of the search depends on the length of the intervals. Assuming that the indices of the k bits found in the first part of the attack are uniformly distributed, the average distance between them is $r = \log p/(k+1)$ bits, so the search of each interval is expected to take about $2^{\log p/(k+1)}$ modular multiplications. There is a trade-off between the data complexity of the attack and the time complexity of the second part. By increasing the data complexity, we expect to find more bits in the first part of the attack (larger k), which allows us to search for the values of fewer bits at a time in the second part, and vice versa.

The attack is presented in Algorithm 11, where Step 1–3 describe the first part of the attack and Step 4 describes the second part.

Using our standard parameters, if we request the decryption of 2^{24} ciphertexts which contain b, then about $2^{24} \cdot 2^{10} \cdot 2^{-27} = 2^7$ of the intermediate values of j are expected to contain a. About half of them are expected to appear in an iteration for which the corresponding bit of d is 1, and therefore $k = 2^6$, and the average length r of the intervals is approximately 2^4 bits. The time complexity of the second part of the attack is thus about $2^{16} \cdot 2^6 = 2^{22}$. Additional 2^{24} ciphertexts are required for Step 2 of the attack, and thus the total data complexity is 2^{25}.

6.2.2. *Known Plaintext Attack*

The following known plaintext bug attack is based on extracting square roots to reverse the LTOR algorithm, and find the bits of the secret exponent from the LSB to the MSB (this is the reverse order of the order in which they are used in the exponentiation algorithm).

The first step of the attack discards all ciphertexts which do not contain the word b, as they are less likely to cause the execution of the buggy instruction. Unlike the chosen ciphertext model, we cannot use the multiplicative properties of the cryptosystem in order

1. Choose 2^{24} random ciphertexts which contain b, and ask for their decryption on the buggy machine.
2. Obtain the correct decryptions of the chosen ciphertexts.
3. For every incorrectly decrypted ciphertext C do
   (a) Let t be the location of b in C.
   (b) Denote the correct decryption of C by M and the incorrect decryption by $\hat{M}$.
   (c) Set $Q = \hat{M}/M$.
   (d) For $j = 0$ to $\log p$ do
      i. For every 2^j-th modular root β of Q modulo p and every $0 \leq s \leq \lfloor \log p/w \rfloor$ do
         A. Compute $X = \delta_{s,t}(\beta - 1)^{-1}C^{-1} \mod p$.
         B. If X contains the word a in location s then set $d_j = 1$, save the tuple (C,X,j) and proceed to the next ciphertext.
4. For every saved tuple (C,X,j) in descending order of j do:
   (a) Complete the unknown values among $d_{\log p}, \ldots, d_{j+2}, d_{j+1}$, such that the intermediate value of z^2 after j iterations of exponentiating C is X.

Algorithm 11: Chosen Ciphertext Attack Against Pohlig–Hellman with LTOR

to identify the incorrect decryptions, and thus we analyze all the remaining ciphertexts and use statistical methods to identify the bits of the secret exponent.

When i ($i \in \{0, 1, \ldots, \log p\}$) least significant bits of the secret exponent d are already known, we can reverse the last i iterations of the LTOR algorithm and compute the value of the variable z after $\log p - i$ iterations of the exponentiation (since this process involves extracting square roots, we get up to $r = \gcd(2^i, p-1)$ candidates for the value of z). We expect that if $d_i = 1$, then for a fraction of 2^{-27} of the ciphertexts the value of z is a result of a buggy multiplication (they form a fraction of $2^{-27}/r$ of all candidates). Also, both in the case of $d_i = 0$ and in the case of $d_i = 1$, a fraction of 2^{-27} of the candidates are values which may be the results of buggy multiplications, with one execution of the buggy instruction. Therefore, if $d_i = 0$ then only a fraction of 2^{-27} of the candidates is expected to be in the set W, while if $d_i = 1$, a fraction of $\left(1 + \frac{1}{r}\right) 2^{-27}$ of the candidates is expected to be in W. In order to distinguish between these two distributions with high probability, $4r^2 2^{27}$ ciphertexts C *that contain* a are sufficient. Since only a fraction of 2^{-27} of all the ciphertexts in the available data are expected to contain a, we require a total of $4r^2 2^{54}$ known ciphertexts for the attack. The attack is presented in Algorithm 12.

7. Discussion on Vulnerabilities of Other Kinds of Schemes

In this section, we consider other schemes that are likely to be vulnerable in the presence of a multiplication bug.

7.1. *Elliptic Curve Schemes*

In cryptosystems based on elliptic curves, exponentiations are replaced by multiplying a point by a constant. For example, an attack on EC-ElGamal would be similar to an

1. Let $r = \gcd(2^i, p-1)$.
2. Discard all the ciphertexts that do not contain the word b (out of the $4r^2 2^{27}$ known messages). They are not used in the attack.
3. Set $d_0 = 1$.
4. For $i = 1$ to $\log p$
   (a) Reverse the last i iterations of the LTOR exponentiation algorithm for all the ciphertexts, and obtain $r = \gcd(2^i, p-1)$ possible values from each ciphertext. Denote the set of all values retrieved by Y.
   (b) Compute $|Y \cap W|$ (using the method described in Sect. 6.2.1).
   (c) If $|Y \cap W| \simeq 2^{-27} |Y|$ set $d_i = 0$.
   (d) If $|Y \cap W| \simeq \left(1 + \frac{1}{r}\right) 2^{-27} |Y|$ set $d_i = 1$.

Algorithm 12: Known Plaintext Attack Against Pohlig–Hellman with LTOR

attack on ElGamal with simple adjustments. 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.

7.2. *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 [18], MARS [9], DFC [13], MultiSwap [25], Nimbus [32] and RC6 [23], the stream cipher Rabbit [7], the message authentication code UMAC [6], 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, which is then used to update the internal state of the cipher. In faulty implementations with word size 8 or 16 (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. This is, however, an unlikely scenario, since bugs in processors with small words are expected to cause frequent errors, and therefore 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	Sects.	Data	Pre-comp. time	Attack time	Complexity for 32-bit words[a]	Complexity for 64-bit words[a]
Pohlig–Hellman (and ElGamal encryption)	RTOL	CP/CC	5.1.1	$2\left\lceil \frac{\log p}{r} \right\rceil$ [b]	$\log p$	$\frac{\log p + 2^r}{r}$ [b]	$2^6/2^{10}/2^{27}$	$2^6/2^{10}/2^{27}$
	RTOL	CP/CC	6.1.1	$\log p$	$\log p$	$\log p$	$2^{10}/2^{10}/2^{10}$	$2^{10}/2^{10}/2^{10}$
	RTOL	KP	6.1.2	$\frac{8 \cdot 2^{2w} \cdot w^2}{\log^2 p}$	–	$\frac{4 \cdot 2^{2w} \cdot w^2}{\log^2 p}$	$2^{57}/-/2^{57}$	$2^{123}/-/2^{123}$
	LTOR	ACP/ACC	4.1.1	$\log p$	–	$\log p$	$2^{10}/-/2^{10}$	$2^{10}/-/2^{10}$
	LTOR	ACP/ACC	4.1.2	$\frac{\log p}{\alpha}$ [c]	c	$\frac{\log p}{\alpha}$ [c]	$2^9+/-/2^9+$	$2^9/-/2^9$
	LTOR	CP/CC	4.1.3	$\frac{2 \cdot 2^w \cdot w}{\log p}$	–	$\frac{2 \cdot 2^w \cdot w}{\log p}$	$2^{28}/-/2^{28}$	$2^{61}/-/2^{61}$
	LTOR	CP/CC	6.2.1	$\frac{4k \cdot 2^w \cdot w}{\log p}$ [d]	–	$\frac{2k \cdot 2^w \cdot w}{\log p} + (k+1)2^{\log p/(k+1)}$ [d]	$2^{25}/-/2^{25}$	$2^{56}/-/2^{56}$
	LTOR	KP	6.2.2	$\frac{4r^2 2^{2w} w^2}{\log^2 p}$ [e]	–	$\frac{4r^2 2^{2w} w^2}{\log^2 p}$ [e]	$r^2 2^{56}/-/r^2 2^{56}$	$r^2 2^{122}/-/r^2 2^{56}$
RSA	CRT	CC	3	1	–	1	1/–/1	1/–/1
	RTOL	CC	5.2.1	$\left\lceil \frac{\log n}{r} \right\rceil$ [b]	$\frac{2^{2w} w^2}{\log^2 n}$	$\frac{\log n + 2^r}{r}$ [b]	$2^5/2^{54}/2^{27}$	$2^5/2^{120}/2^{27}$
	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 with OAEP	RTOL	ACC	5.3.1	$\log n$	–	$\frac{2^{2w} \cdot w^2}{\log n}$	$2^{10}/-/2^{64}$	$2^{10}/-/2^{130}$
	LTOR	ACC	4.3.1	$\log n$	–	$\frac{2^{2w} \cdot w^2}{\log n}$	$2^{10}/-/2^{64}$	$2^{10}/-/2^{130}$
	LTOR	CC	4.3.2	$\frac{4 \cdot 2^w \cdot w}{\log n}$	$\frac{4 \cdot 2^{2w} \cdot w^2}{\log^2 n}$	$\frac{4 \cdot 2^w \cdot w}{\log n}$	$2^{29}/2^{56}/2^{29}$	$2^{62}/2^{122}/2^{62}$

w is the word size (in bits) of the faulty processor

KP Known plaintext, *CP* chosen plaintext, *ACP* adaptive chosen plaintext, *CC* chosen ciphertext, *ACC* adaptive chosen ciphertext

[a] Complexity is described in terms of data/pre-computation time/attack time, assuming $\log p = \log n = 1024$

[b] r is a parameter of the attack. The presented numbers are for $r = 2^5$

[c] $\alpha \in [1.5, 2]$ is a constant, which can be increased (within this range) by investing a longer pre-computation time

[d] k is a parameter of the attack. The presented numbers are for $k = 2^6$

[e] In the context of this attack $r = \gcd\left(2^{\lfloor \log p \rfloor + 1}, p - 1\right)$

8. Summary and Countermeasures

We presented several attacks against exponentiation based public-key and secret-key cryptosystems, including Pohlig–Hellman, RSA, and ElGamal encryption. We described such attacks for the two most common implementations of exponentiation. We also discussed the possible 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 Sects. 4.3 and 5.3, and unlike the case of fault attacks, the mere knowledge that an error 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 (the techniques for blinding RSA are based on [10]). 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 at all. Another alternative is to exponentiate modulo $n \cdot r$, where r is a small (e.g., 32 bit) prime unknown to the attacker, and reduce the result mod n only at the last step [26].

Acknowledgements

The authors would like to thank Orr Dunkelman for his comments. We also thank the anonymous referees for their valuable comments and suggestions which improved the results of this paper. The first two authors were supported in part by the Israel MOD Research and Technology Unit.

Appendix: Brief Descriptions of Several Cryptosystems

The Pohlig–Hellman Cryptosystem and Pohlig–Hellman-Shamir Protocol

The Pohlig–Hellman cryptosystem [22] 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 $c = m^e \mod p$. Bob can decrypt c by computing its e-th root modulo p. In practice, the decryption is performed by computing $c^d \mod p$, 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 [27] 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 $c_1 = m^{e_A} \mod p$. Bob then computes $c_2 = c_1^{e_B} \mod p$ and sends it back to Alice. Alice decrypts c_2 and sends the decryption $c_3 = c_2^{d_A} \mod p$ to Bob. Finally, Bob decrypts c_3 to get the message $m = c_3^{d_B} \mod p$. The protocol is secure under standard computational assumptions (the Diffie–Hellman assumption), but not against man in the middle attacks.

The RSA Cryptosystem

RSA [24] 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 $c = m^e \mod n$. 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.

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 CRT to compute the decryption $c^d \mod n$. Such an implementation speeds up the decryption by a factor of four 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: $m_p = c^{d_p} \mod p$, and $m_q = c^{d_q} \mod q$, where $d_p = d \mod p-1$ and $d_q = d \mod q-1$. 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 $m = (xm_p + ym_q) \mod n$, where x and y are pre-computed integers that satisfy:

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

OAEP

Optimal Asymmetric Encryption Padding (OAEP) [4] and OAEP+ [28] 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

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

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.

Known Hardware Bugs

In this appendix, we give a partial list of known hardware bugs. A quick Internet search yields many more bugs, some of which were never officially acknowledged by the manufacturers. It is safe to assume that hardware manufacturers are aware of many more bugs which were never made publicly known or which were later corrected by firmware updates, and that there are many more hardware bugs waiting to be discovered.

- Pentium FDIV bug [14,15]:
  This well-known bug in the FDIV instruction of the Pentium processor was caused by missing entries in a lookup table. These entries were omitted due to a programming error. The bug caused inaccurate results in floating point division for some of the inputs. Byte magazine [14] assessed that the bug influenced about one in a billion floating point divisions (for random inputs).
- Intel Core 2 TLB bug [16]:
  Intel Core 2 memory management unit has a reported error in the translation lookaside buffer (TLB—a unit responsible for translating virtual memory addresses to physical addresses). Global entries in the TLB may not be invalidated when the table

is initialized, which may cause the processor to read data from incorrect memory addresses. This bug may cause the system to stop responding or crash.

– AMD Phenom 9700/TLB system lockup bug [31]:
Before its release, AMD found that the Phenom 9700 quad-core processor had a TLB bug which may cause the CPU to hang when all four cores are running at full load. The discovery of this bug caused AMD to delay the release of this model.
– Intel 80286 popf bug [21]:
A bug in the popf instruction (which pops the flags off the stack) allowed interrupts to be executed, even when they were supposed to be disabled. This bug is an example of a very simple instruction which changes the state of the CPU even when no change is needed.
– Intel Pentium f00f bug [17]:
Under certain conditions, when a program tried to execute a specific invalid opcode, the entire system would hang instead of generating an "invalid opcode exception" (which would terminate the errant program).
– AMD Athlon/Duron with AGP bug [1]:
This memory management bug caused Linux systems to hang when the system displayed AGP graphics. The bug was caused because of improper handling of extended paging (which supported large page sizes).
– MOS Technology 6502 bugs[33]:
The 6502 model of the MOS processor introduced binary coded decimal (BCD) instructions for manipulating decimal numbers without first converting them to binary. If a hardware interrupt occurred when the processor was in BCD mode, it would not revert back to binary mode for the execution of the interrupt handler. Another bug in this processor caused the JMP instruction to read its destination address from the wrong memory address under certain conditions.
– Cyrix coma bug [3]:
The bug in the Cyrix 6x86 series could cause the processor to stop responding to interrupts while executing an infinite loop. Because interrupts were ignored, there was no way to abort the loop, and the system would stop responding.
– Intel 80386 multiplication bug [20]:
The first x86 with 32-bit architecture exhibited a bug in its 32-bit multiplication instruction. The bug may have caused the processor to stop responding. Even after the discovery of the bug, the buggy processors continued to be sold as "16 BIT S/W ONLY".
– Intel Pentium Pro and Pentium II FPU bug [11]:
This bug (regarded by Intel as the "flag erratum") caused unexpected behavior when trying to convert from floating point to integer, if the result was too large to fit in an integer variable.

References

[1] AMD, *Linux Kernel Issue with Systems Using AGP Graphics—Application Note*, August 2002. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/26698.pdf

[2] J. Appelbaum, J. Horchert, C. Stöcker, *Shopping for Spy Gear: Catalog Advertises NSA Toolbox*, Der Spiegel, 29 December 2013. Online edition: http://www.spiegel.de/international/world/catalog-reveals-nsa-has-back-doors-for-numerous-devices-a-940994.html
[3] A.D. Balsa, *The Cyrix 6x86 Coma Bug*. http://www.tux.org/~balsa/linux/cyrix/index.html
[4] M. Bellare and P. Rogaway, *Optimal Asymmetric Encryption—How to Encrypt with RSA (Extended Abstract), Advances in Cryptology, Proceedings of EUROCRYPT'94, LNCS 950* (Springer, Berlin, 1995), pp. 92–111
[5] E. Biham, Y. Carmeli, A. Shamir, Bug attacks, in *Advances in Cryptology, Proceedings of CRYPTO'08, LNCS 5157* (Springer, Berlin, 2008) pp. 221–240.
[6] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, P. Rogaway, UMAC: fast and secure message authentication, in *Advances in Cryptology, Proceedings of CRYPTO'99, LNCS 1666* (Springer, Berlin, 1999) pp. 215–233.
[7] M. Boesgaard, M. Vesterager, T. Pedersen, J. Christiansen, O. Scavenius, Rabbit: a new high performance stream cipher, in *Proceedings of Fast Software Encryption 10, LNCS 2887* (Springer, Berlin, 2004) pp. 307–329.
[8] D. Boneh, R.A. DeMillo, R.J. Lipton, On the importance of checking cryptographic protocols for faults, in *Advances in Cryptology, Proceedings of EUROCRYPT'97, LNCS 1233* (Springer, Berlin, 1997) pp. 37–51.
[9] C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S.M. Matyas Jr., L. O'Connor, M. Peyravian, D. Safford, N. Zunic, MARS: a candidate cipher for AES, in *AES—The First Advanced Encryption Standard Candidate Conference, Conference Proceedings, 1998*.
[10] D. Chaum, Blind signatures for untraceable payments, in *Advances in Cryptology, Proceedings of CRYPTO'82* (Plenum Press, Berlin, 1983) pp. 199–203.
[11] R.R. Collins, *Inside the Pentium II Math Bug*, Dr. Dobb's Portal, August 1997. http://www.ddj.com/184410254
[12] T. ElGamal, *A public key cryptosystem and a signature scheme based on discrete logarithms*. IEEE Trans. Inf. Theory **31**(4), 469–472 (1985).
[13] H. Gilbert, M. Girault, P. Hoogvorst, F. Noilhan, T. Pornin, G. Poupard, J. Stern, S. Vaudenay, Decorrelated fast cipher: an AES candidate, in *AES—The First Advanced Encryption Standard Candidate Conference, Conference Proceedings, 1998*.
[14] T.R. Halfhill, The truth behind the Pentium bug, in *BYTE Magazine*, March 1995. http://www.byte.com/art/9503/sec13/art1.htm
[15] Intel, *FDIV Replacement Program—Statistical Analysis of Floating Point Flaw: Intel White Paper*, July 2004. http://support.intel.com/support/processors/pentium/sb/CS-013007.htm
[16] Intel, *Intel® Core™2 Duo Processor E8000 and E7000 Series*, July 2004. http://www.intel.com/design/processor/specupdt/318733.pdf
[17] Intel, *Intel® Processor—Invalid Instruction Erratum Overview*, November 1997. http://www.intel.com/support/processors/pentium/ppiie/
[18] X. Lai and J.L. Massey and S. Murphy, Markov ciphers and differential cryptanalysis, in *Advances in Cryptology, Proceedings of EUROCRYPT'91, LNCS 547* (Springer, Berlin, 1992) pp. 17–38.
[19] A.J. Menezes, P.C. van Oorschot and S.A. Vanstone, *Handbook of Applied Cryptography* (CRC Press, Boca Raton, 1996).
[20] S. Mueller *Upgrading and Repairing PCs*, Eighth edition, Que Publishing, 1998. http://www.informit.com/content/downloads/que/upgrading/fourteenth_edition/DVD/PCs8th.pdf
[21] L. Osterman, *Remembering Old CPU Bugs*, Larry Osterman's WebLog, February, 2007. http://blogs.msdn.com/larryosterman/archive/2007/02/06/remembering-old-cpu-bugs.aspx
[22] S.C. Pohlig, M.E. Hellman, *An improved algorithm for computing logarithms over GF(p) and its cryptographic significance*. IEEE Trans. Inf. Theory **24**(1), 106–111 (1978).
[23] R.L. Rivest, M.J.B. Robshaw, R. Sidney, and Y.L. Yin, The RC6 block cipher, in *AES—The First Advanced Encryption Standard Candidate Conference, Conference Proceedings, 1998*.
[24] R.L. Rivest, A. Shamir, L. Adleman, *A method for obtaining digital signatures and public-key cryptosystems*. Commun. of the ACM **21**(2), 120–126 (1978).
[25] B. Screamer, *Microsoft's Digital Rights Management Scheme—Technical Details*, October 2001. http://cryptome.org/ms-drm.htm
[26] A. Shamir, *RSA for paranoids*. CryptoBytes **1**(3), 1–4 (1995).

[27] A. Shamir, R.L. Rivest, L.M. Adleman, *Mental poker,* in D.A. Klarner (ed.), *The Mathematical Gardner* (Wadsworth, Belmont, 1981) pp. 37–43.
[28] V. Shoup, *OAEP Reconsidered (Extended Abstract), Advances in Cryptology, Proceedings of CRYPTO 2001, LNCS 2139* (Springer, Berlin, 2001) pp. 239–259.
[29] S. Staff, *Inside TAO: Documents Reveal Top NSA Hacking Unit*, Der Spiegel, 29 December 2013. Online edition: http://www.spiegel.de/international/world/the-nsa-uses-powerful-toolbox-in-effort-to-spy-on-global-networks-a-940969-3.html
[30] U.S.D. of Defense, *Defense Science Board Tas Force on High Performance Microchip Supply*, February 2005. http://www.acq.osd.mil/dsb/reports/2005-02-HPMS_Report_Final.pdf
[31] Theo Valich, AMD delays Phenom 2.4 GHz due to TLB errata in *The Inquirer*, November 2007. http://www.theinquirer.net/gb/inquirer/news/2007/11/18/amd-delays-phenom-ghz-due-tlb
[32] A. Warner Machado, The Nimbus cipher: a proposal for NESSIE, in *NESSIE Proposal*, September 2000.
[33] Wikipedia, *MOS Technology 6502*. http://en.wikipedia.org/wiki/MOS_Technology_6502