Fault-Based Attack of RSA Authentication

Andrea Pellegrini, Valeria Bertacco and Todd Austin
University of Michigan
{*apellegrini, valeria, austin*}*@umich.edu*

ABSTRACT

For any computing system to be secure, both hardware and software have to be trusted. If the hardware layer in a secure system is compromised, not only it would be possible to extract secret information about the software, but it would also be extremely hard for the software to detect that an attack is underway. In this work we detail a complete end-to-end fault-attack on a microprocessor system and practically demonstrate how hardware vulnerabilities can be exploited to target secure systems. We developed a theoretical attack to the RSA signature algorithm, and we realized it in practice against an FPGA implementation of the system under attack. To perpetrate the attack, we inject transient faults in the target machine by regulating the voltage supply of the system. Thus, our attack does not require access to the victim system's internal components, but simply proximity to it.

The paper makes three important contributions: first, we develop a systematic fault-based attack on the modular exponentiation algorithm for RSA. Second, we expose and exploit a severe flaw on the implementation of the RSA signature algorithm on OpenSSL, a widely used package for SSL encryption and authentication. Third, we report on the first physical demonstration of a fault-based security attack of a complete microprocessor system running unmodified production software: we attack the original OpenSSL authentication library running on a SPARC Linux system implemented on FPGA, and extract the system's 1024-bit RSA private key in approximately 100 hours.

1. INTRODUCTION

Public-key cryptography schemes (Figure 1.a) are widely adopted wherever there is a need to secure or authenticate confidential data on a public communication network. When deployed with sufficiently long keys, these algorithms are believed to be unbreakable. Strong cryptographic algorithms were first introduced to secure communications among high performance computers that required elevated confidentiality guarantees. Today, advances in semiconductor technology and hardware design have made it possible to execute these algorithms in reasonable time even on consumer systems, thus enabling the mass-market use of strong encryption to ensure privacy and authenticity of individuals' personal communications. Consequently, this transition has enabled the proliferation of a variety of secure services, such as online banking and shopping. Examples of consumer electronics devices that routinely rely on high-performance public key cryptography are Blu-ray players, smart phones, and ultra-portable devices. In addition, low-cost cryptographic engines are mainstream components in laptops, servers and personal computers. A key requirement for all these hardware devices is that they must be affordable. As a result, they commonly implement a straightforward design architecture that entails a small silicon footprint and low-power profile.

Our research focuses on developing an effective attack on mass-market crypto-chips. Specifically, we demonstrate an effective way to perpetrate fault-based attacks on a microprocessor system in order to extract the private key from the cryptographic routines that it executes. Our work builds on a theoretical fault-based attack proposed in [6], and extends it to stronger implementations of the RSA-signature algorithm. In addition, we demonstrate the attack in practice by generating a number of transient faults on an FPGA-based SPARC system running Linux, using simple voltage manipulation, and applying our proposed algorithm to the incorrectly computed signatures collected from the system under attack. This attack model is not uncommon since many embedded systems, for cost reasons, are not protected against enviromental manipulations. Our fault-based attack can be successfully perpetrated also on systems adopting techniques such as hardware self-contained keys and memory/bus encryption.

The attack requires only limited knowledge of the victim system's hardware. Attackers do not need access to the internal components of the victim chip, they simply collect corrupted signature outputs from the system while subjecting it to transient faults. Once a sufficient number of corrupted messages have been collected, the private key can be extracted through offline analysis.

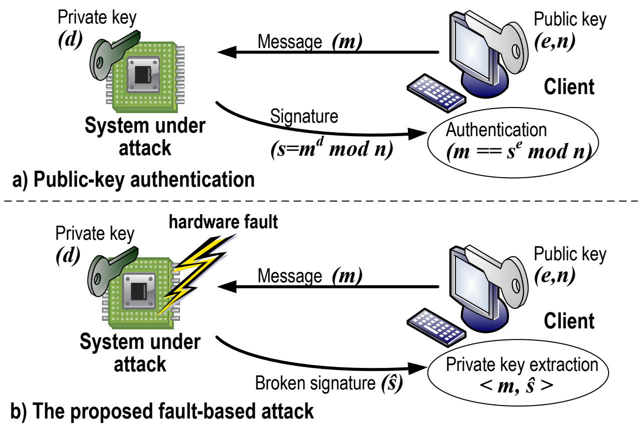

Figure 1: Overview of public key authentication and our fault-based attack. a) in public key authentication, a client sends a unique message m to a server, which signs it with its private key d. Upon receiving the digital signature s, the client can authenticate the identity of the server using the public key (n, e) to verify that s will produce the original message m. b) Our fault-based attack can extract a server's private key by injecting faults in the server's hardware, which produces intermittent computational errors during the authentication of a message. We then use our extraction algorithm to compute the private key d from several unique messages m and their corresponding erroneous signatures $\hat{s}$.

Occurrence of hardware faults. Current silicon manufacturing technology has reached such extreme small scales that the occurrence of transient hardware failures is a natural phenomenon, caused by environmental alpha particles or neutrons striking switching transistors. Similarly, occasional transient errors can be induced by forcing the operative conditions of a computer system. A systematic vulnerability to these attacks can also be introduced during the manufacturing process, by making some components in the system more susceptible to transient faults than others.

Several consumer electronic products, such as ultra-mobile computers, mobile phones and multimedia devices are particularly susceptible to fault-based attacks: it is easy for an attacker to gain physical access to such systems. Furthermore, even a legitimate user of a device could perpetrate a fault-based attack on it to extract confidential information that a system manufacturer intended to keep secure (as, for instance, in the case of multimedia players).

Contributions of this work. This paper presents a fault-based technique to perpetrate an attack on RSA authentication by exploiting microarchitectural or circuit-level vulnerabilities in digital hardware devices. It makes three key contributions: first, we extend the theoretical work proposed by Boneh *et al.*, in [6] and develop a novel RSA authentication attack (see also Figure 1.b), which extracts a server's RSA private key by extracting information through perturbing the fixed-width modular exponentiation algorithm used in the popular OpenSSL library [1]. OpenSSL is an open-source secure sockets layer (SSL) implementation of RSA authentication [13], widely deployed in internet and web security applications, including the Apache web server, BIND DNS server and the OpenSSH secure shell. The second contribution is the discovery of a severe vulnerability in the software implementation of RSA authentication in OpenSSL, which can be expoited to perform fault-based attacks.

Finally, we apply our technique to demonstrate the fault-based attack on a SPARC-based microprocessor system, implemented on FPGA and running Linux. We inject faults into the system through by simply manipulating the voltage supply, resulting in occasional transient faults in the SPARC processor's multiplier. The injected faults create computation errors in the system's RSA authentication routines, which we exploit to extract the private key. The attack is perpetrated on an unmodified OpenSSL (version 0.9.8i). In our experiment we show that we can fully extract the server's 1024-bit private key in approximately 100 hours. Once the machine's private key is acquired, it becomes possible for the attacker to pose as the compromised server to unsuspecting clients.

It is worth noting that this attack is immune to protection mechanisms such as system bus and/or memory encryption, and that it does not damage the device, thus no tamper evidence is left to indicate that a system has been compromised.

2. RELATED WORK

Several algorithms have been proposed to implement the exponentiation of large numbers, including techniques based on the *Chinese Remainder Theorem (CRT)*. This algorithm is particularly prone to fault attacks, and several of them have been suggested as reported in the literature [6, 10, 15]. Other algorithms for exponentiation, such as *square-and-multiply* and *right-to-left binary exponentiation*, are also susceptible to fault-based attacks [6]. Each uses an ad-hoc fault model, ranging from altering the private exponent stored in the system [3], to injecting single-bit errors into those registers storing partial exponentiation results [6], to carefully timing fault-injections to corrupt a specific operation within the exponentiation, as theorized in [7]. Our theoretical contribution adopts the same single-bit flip fault model proposed in [6].

The OpenSSL library quickly computes RSA private key signatures using a CRT-based algorithm, and then checks the correctness of the generated result (detecting potential attacks) by verifying it with the public key and comparing the result with the original message. If a mismatch is observed, it resorts to the more time consuming *left-to-right squaring* as a safety measure, since this latter algorithm is considered resilient to security attacks. In our work we rely on single-bit faults to attack precisely *left-to-right squaring* (shown in Figure 2), since this algorithm is considered a "safe back-up" in the OpenSSL library. While *left-to-right squaring* is algorithmically similar to *right-to-left repeated squaring*, single-bit faults have a distinctly different impact on the computational results. This paper presents the first systematic approach to fault-based attacks of the *left-to-right squaring* algorithm, used in the popular OpenSSL cryptographic library. We will refer to the particular implementation of the *left-to-right exponentiation* deployed in OpenSSL as *Fixed Window Exponentiation (FWE)*.

A theoretical example of a similar attack is presented in [5], where functional errors in the hardware executing the exponentiation algorithm are used to break RSA and other strong cryptographic systems. In that work, the authors indicate how a functional bug in the multiplier of a microprocessor can be exploited to this end. Note, however, that the attack proposed is viable only if the needed bug was to escape the hardware verification phase, which is a highly improbable proposition, given the extreme effort dedicated to modern designs' validation [9].

The number of reports that detail actual physical implementations of these attacks perpetrated through erroneous computation in the hardware layer is very scarce. Recently, an attack on a physical implementation of the *square-and-multiply* algorithm running on a microcontroller was demonstrated in [14]. Faults injected in the microcontroller were used to control the program counter of the victim, so that the program executing the exponentiation algorithm would some specific instructions. Additionally, a few other theoretical attacks have been physically demonstrated on simple microcontroller-based systems and smart cards [2, 4]. One of our key contributions in this paper is the first physical demonstration of a fault-based attack on a complete microprocessor-based system, running unmodified software, including the Linux operating system and a current version of the OpenSSL library.

3. AUTHENTICATION WITH RSA

RSA is a commonly adopted public key cryptography algorithm [13]. Since it was introduced in 1977, RSA has been widely used for establishing secure communication channels and for authenticating the identity of service providers over insecure communication mediums. In the authentication scheme, the server implements *public key authentication* with clients by signing a unique message from the client with its private key, thus creating what is called a *digital signature*. The signature is then returned to the client, which verifies it using the server's known public key (see also Figure 1.a).

The procedure for implementing public key authentication requires the construction of a suitable pair of $public\ key\ (n, e)$ and $private\ key\ (n, d)$. Here n is the product of two distinct big prime numbers, and e and d are computed such that, for any given message m, the following identity holds true: $m \equiv (m^d)^e \bmod n \equiv (m^e)^d \bmod n$. To authenticate a message m, the server attaches a signature s to the original message and transmits the pair. The server generates s from m using its private key with the following computation: $s \equiv m^d \bmod n$. Anyone who knows the public key associated with the server can then verify that the message m and its signature s were authentic by checking that: $m \equiv s^e \bmod n$.

3.1 Fixed-window modular exponentiation

Modular exponentiation ($m^d \bmod n$) is a central operation in public key cryptography. Many cryptographic schemes, including RSA, ElGamal, DSA and Diffie-Hellman key exchange, heavily rely on modular exponentiation for their algorithms. Several algorithms that implement modular exponentiation are available [11]. In this paper we focus on the fixed window exponentiation (FWE) algorithm ([11] - chapter 14). This algorithm, used in OpenSSL-0.9.8i, is guaranteed to compute the modular exponentiation function in constant time, and its performance depends only on the length of the exponent. Because of this reason, the algorithm is impervious to timing-based attacks [8].

The fixed-window modular exponentiation algorithm is very similar to square-and-multiply [14], but instead of examining each individual bit of the exponent, it defines a window, w bits wide, and partitions the exponent in groups of w bits. Conceptually, the length of the algorithm's window may be either variable or fixed. However, using variable window lengths makes the computation susceptible to timing-based attacks. To avoid these attacks, thus OpenSSL utilizes a fixed window size.

The FWE algorithm operates by computing the modular exponentiation for each window of w bits of the exponent and accumulating the partial results. Since w typically comprises just a few bits, the exponent is correspondingly a small number, between 0 and $(2^w - 1)$, leading to a practical computation time. Figure 2 reports the pseudo-code for the algorithm, where an accumulator register `acc` stores the partial results. The algorithm starts from the most significant bits of the exponent d and, during each iteration, the bits of d corresponding to the window under consideration are extracted and used to compute $m^{d[win\_idx]} \bmod n$ (lines 7-9). In addition, the bits of the window of d under consideration must be shifted by w positions. Since d is the exponent of the message, shifting d to the left by one position corresponds to squaring the base. Shifting is thus accomplished by squaring the accumulator w times (lines 5-6). Once all windows of size w have been considered, the accumulator contains the final value of $m^d \bmod n$. Note that, in practice, the powers of m from 0 to $2^w - 1$ are pre-computed and stored aside, so that line 9 in the code reduces to a simple lookup and multiplication. By leveraging the pre-computed powers of m, the algorithm only requires a constant number of multiplications.

It is possible to reduce the window size w down to 1, in which case the FWE algorithm degrades into square-and-multiply. However, using larger values of w brings noticeable benefits to the computation time, because of the smaller number of multiplications required. Finally, if we define k as the ratio between the number of bits in d and w: $k = \#\text{bits}(d)/w$, the general expression computed by the FWE algorithm is:

$$\begin{aligned} s &= (\cdot\cdot(m^{d_{k-1}})^{2^w})\cdots m^{d_i})^{2^w})\cdots m^{d_1})^{2^w})m^{d_0} \bmod n \\ &= m^{d_{k-1}2^{w(k-1)}}\cdots m^{d_i 2^{wi}}\cdots m^{d_1 2^w} m^{d_0} \bmod n \end{aligned} \quad (1)$$

```
1  FWE(m, d, n, win_size)
2     num_win = #bits(d) / win_size
3     acc = 1
4     for(win_idx in [num_win-1..0] )
5        for(sqr_iter in [0..win_size-1] )
6            acc = (acc * acc) mod n
7        d[win_idx] =
8            bits(d, win_idx*win_size,win_size)
9        acc = (acc * m^d[win_idx]) mod n
10    return acc
```

Figure 2: Fixed window exponentiation. The algorithm computes $m^d \bmod n$. For performance, the exponent d is partitioned in `num_win` windows of `win_size` bits. Moreover, to ensure a constant execution time, independent from the specific value of the exponent d, a table containing all the powers of m from 0 to $2^{win\_size} - 1$ is precomputed and stored aside.

4. HARDWARE FAULT MODEL

The fault-based attack that we developed in this work exploits hardware faults injected at the server side of a public key authentication (see Figure 1.b). Specifically, we assume that an attacker can occasionally inject faults that affecting the result of a multiplication computed during the execution of the fixed-window exponentiation algorithm. Consequently, we assume that the system is subjected to a battery of infrequent short-duration transient faults, that is, faults whose duration is less than one clock cycle, so that they impact at most one multiplication during the entire execution of the exponentiation algorithm. Moreover, we only consider hardware faults that produce a multiplication result differing from the correct one in only one bit position, and simply disregard all others.

To make this attack possible, faults with the characteristics described must be injected in the attacked microprocessor. For this purpose, we exploit a circuit-level vulnerability common in microprocessor design: multiplier circuits tend to be fairly complex, and much effort has been dedicated to developing high performance multipliers, that is, multipliers with short critical path delays. Even so, often the critical path of a microprocessor system goes through the multiplier circuit [12]. If environmental conditions (such as high temperatures or voltage manipulation by an attacker) slow down the signal propagation in the system, it is possible that signals through the critical path do not reach their corresponding registers or latches before the next clock cycle begins. In such situations, one of the first units to fail in computing correct results tends to be the multiplier, because its "margin" of delay is minimal. Note that not all multiplications would be erroneous, only those which required values generated through the critical path.

In order to perpetrate our attack, we collect several pairs of messages m and their corrupted signatures $\hat{s}$, where $\hat{s}$ has been subjected to only one transient fault with the characteristics described. In Section 6.1 we show how we could inject faults with the proper characteristics in the authenticating machine. Moreover, while our attack requires a single fault placed in the exponentiation multiplication operation, it is resilient to multiple errors and errors placed in other operations; however, those will not yield any useful information about the private key.

4.1 FWE in presence of transient faults

The fixed-window exponentiation algorithm in the OpenSSL library does not validate the correctness of the signature produced before sending it to the client, a vulnerability that we exploit in our attack. We now analyze the impact of a transient fault on the output of the FWE algorithm (see Section 3.1). As mentioned above, the software-level perception of the fault is a single-bit flipped in one of the multiplications executed during FWE. With reference to Figure 2, during FWE, multiplications are computed executing during accumulator squaring (line 6), message window exponentiation (line 9). For sake of simplicity, in this analysis we only consider messages that have been hit by a fault during any of the accumulator squaring multiplications of line 6, the reasoning extends similarly for faults affecting the multiplications of line 9.

Since the error manifests as a single-bit flip, the corrupted result will be modified by $\pm 2^f$, where f is the position of the bit flipped in the partial result, that is, the location of the corrupted bit f is in the range $0 \leq f < \#\text{bits}(\texttt{acc})$. The error amount is added or subtracted, depending on the transition induced by the flip: if the fault modified a bit from 1 to 0, the error is subtracted, otherwise it is added. Thus, with reference to Eq. (1), showing the computation executed by the FWE algorithm, if a single-bit flip fault hits the server during the pth squaring operation in the computation for the ith window of the exponent d, the system will generate a corrupted signature $\hat{s}$ as follows (the mod n notation has been omitted):

$$\hat{s} = (\cdot\cdot(m^{d_{k-1}})^{2^w})\cdots m^{d_i})^{2^p} \pm 2^f)^{2^{w-p}})\cdots m^{d_1})^{2^w})m^{d_0} \quad (2)$$

or, equivalently,

$$\hat{s} = \left((\prod_{j=i+1}^{k-1} m^{d_j 2^{(j-i)w}}) m^{d_i 2^p} \pm 2^f \right)^{2^{iw-p}} \prod_{j=0}^{i-1} m^{d_j 2^{jw}} \quad (3)$$

5. FAULT-BASED ATTACK TO FWE

In this section we show how to extract the private key in a public key authentication system from a set of messages m and their erroneously signed counterpart $\hat{s}$, which have been collected by injecting transient faults at the server.

We developed an algorithm whose complexity is only polynomial on the size of the private key in bits. The algorithm proceeds by attempting to recover one window of w bits of the private key d at a time, starting from the most significant set of bits. When the first window has been recovered, it moves on to the next one, and so on. While working on a window i, it considers all message-corrupted signature pairs, $< m, \hat{s} >$, one at a time, and attempts to use them to extract the bits of interests. Pairs for which a fault has been injected in a bit position within the window i can be effective in revealing those key's bits. All other pairs will fail at the task, they will be discarded and used again when attempting to recover the next windows of private key bits. The core procedure in the algorithm, applied to one specific window of bits i and one specific $< m, \hat{s} >$ pair, is a search among all possible fault locations, private key window values and timing of the fault, with the goal of finding a match for the values of the private key bits under study. In the next section we present the details of the extraction algorithm.

5.1 Algorithm for private key recovery

THEOREM 5.1. *Given a public key authentication system, $< n, d, e >$ where n and e are known and d is not known, and for which the signature with the private key d of length N is computed using the fixed-window exponentiation (FWE) algorithm with a window size w, we call k the number of windows in the private key d, that is, $k = N/w$. Let us call $\hat{s}$ a corrupted signature of the message m computed with the private key d. Assume that a single-bit binary value change has occurred at the output of any of the squaring operations in FWE during the computation of $\hat{s}$. An attacker that can collect at least $S = k \cdot ln(2k)$ different pairs $< m, \hat{s} >$ has a probability $pr = 1/2$ to recover the private key d of N bits in polynomial time - $\mathcal{O}(2^w N^3 S)$.*

The proof of Theorem 5.1 is presented in Appendix A. We developed an algorithm based on the construction presented there that iterates through all the windows, starting from the one corresponding to the most significant bits. For each window, it considers one message - signature $< m, \hat{s} >$ pair at a time, discarding all of those that lead to 0 or more than one solution for the triplet $< d_i, f, p >$. As soon as a signature is found that provides a unique solution, the value d_i can be determined, and the algorithm can advance to recover the next window of bits.

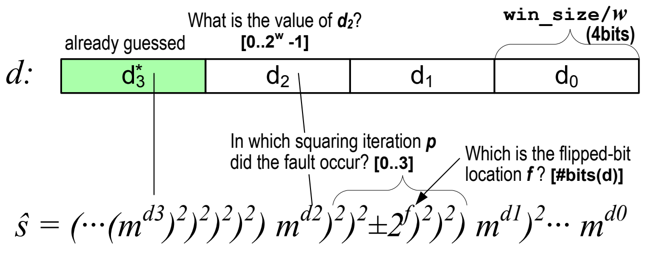
What is the value of d2?
[0..2^w -1]
already guessed
win_size/w
(4bits)
d:
d3*
d2
d1
d0
In which squaring iteration p did the fault occur? [0..3]
Which is the flipped-bit location f ? [#bits(d)]
ŝ = (···(m^d3)^2)^2)^2)^2) m^d2)^2)^2±2^f)^2)^2) m^d1)^2··· m^d0

Figure 3: Example of our private key recovery. The schematic shows a situation where the private key d to be recovered has size 16 bits, and each window is 4 bits long. Key recovery proceeds by determining first the 4 most significant bits in d, d_3. Then in attempting to recover d_2, all possible values for d_2, p and f must be checked to evaluate if they correspond to the signature $\hat{s}$. d_2 may assume values $[0, 15]$, p $[0, 3]$ and f $[0, 15]$.

As an example, consider a window w of size 4, and m and d of 16 bits. Figure 3 illustrates this scenario. Assume that the most significant window has already been identified to be the 4-bit value d_3^*. In the inductive step we must search for an appropriate value of d_2, f and p that satisfy Eq. (10) in the Appendix. The figure shows how the three components of the triplets correspond to different variable aspects of the faulty signature $\hat{s}$.

The core function of the algorithm considers one message and its corresponding signature, and it attempts to determine a valid triplet satisfying Eq. (10). The function is illustrated in the pseudo-code of Figure 4.

```
window_search (m, s, e, win_size, win_idx)
   found = 0;
   for(d[win_idx] in [0..2^win_size-1];
      sqr_iter in [0..win_size-1];
      fault in [0..#bits(d)-1] )
         found += test_equation_10( m, s, e,
           win_idx, d[win_idx], sqr_iter, fault_loc)
   if (found == 1) return d[win_idx]
   else return -1
```

Figure 4: Private key window search. The core function of the private key recovery algorithm considers one message-signature pair and scans through all possible values in the window `d[win_idx]`, the fault location `fault` and the squaring iteration `sqr_iter`. If one and only one solution is found that satisfies Eq. (10), the function returns the value determined for `d[win_idx]`.

The private key recovery algorithm invokes `window_search()` several times: for each window of the private key d, this core function is called using different $< m, \hat{s} >$ pairs, until a successful d_i is obtained. Figure 5 shows the pseudo-code for the overall algorithm. Note that it is possible that no $< m, \hat{s} >$ pair leads to revealing the bits of the window under consideration. In this situation, the algorithm can still succeed by moving on to the next window and doubling the window size. This is a backup measure with significant impact on the computation time. Alternatively it is also possible to collect more $< m, \hat{s} >$ pairs.

The private key extraction algorithm may be optimized in several ways. It is possible to parallelize the computation by distributing the search for a given window over several processes, each attempting to validate the same triplets of values over different signatures. In addition, it is also possible to distribute different values for the candidate triplets over different machines.

```
private_key_recovery ( array<m,s>, e, win_size)
   num_win = #bits(d) / win_size
   for(win_idx in [num_win-1..0] )
      for (<m,s> in array<m,s>)
         d[win_idx] = window_search(m,s,e,
                          win_size, win_idx)
         if (d[win_idx] >= 0) break
      if (d[win_idx] < 0) double_win_size
```

Figure 5: Private-key recovery algorithm. The recovery algorithm sweeps all the windows of the private key, from the most significant to the least one. For each windows it determines the corresponding bits of the private key d by calling `window_search()` until a successful value is returned. If no signature s can be used to reveal the value of `d[win_idx]`, the window size is doubled for the next iteration.

6. EXPERIMENTAL RESULTS

In this section we detail the physical attack that we performed on a SPARC-based Linux system, and analyze the behavior of the system under attack. The device under attack is a complete system mapped on a field-programmable gate array (FPGA) device.

The hardware consists of a SPARC-based Leon3 SoC from Gaisler Research, which is representative of an off-the-shelf commericial embedded device. In our experiments, the unmodified VHDL of the Leon3 was mapped on a Xilinx Virtex2Pro FPGA. The system runs a Debian/GNU distribution with Linux Kernel version 2.6.21 and OpenSSL version 0.9.8i

6.1 Induced fault rate

As we mentioned in Section 4, voltage regulation is critical to an efficient implementiation of a fault-based attack. If the voltage is too high, the rate of faults is too low, and it will require a long time to gather a sufficient number of faulty digital signatures. If the voltage is too low, the fault rate increases, causing system instability and multiple bit errors for each FWE algorithm invocation, thus yielding no private key information.

Figure 6 shows the injected fault rate as a function of the supply voltage. We studied the behavior of the hardware system computing the functions used in the OpenSSL library while being subjected to supply voltage manipulation. In particular, we studied the behavior of the routine that computes the multiplication using 10,000 randomly generated operand pairs of 1,024 bits in length.

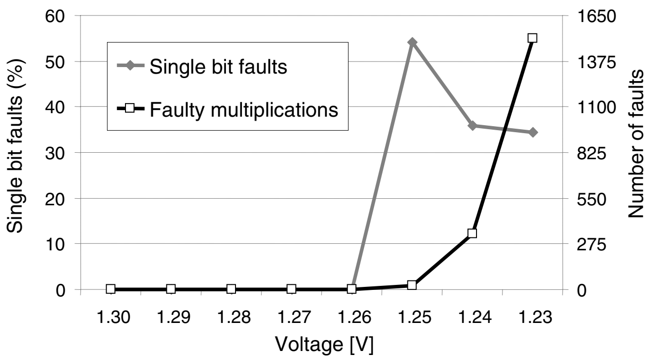

Figure 6: Sensitivity of multiplications executed in OpenSSL to voltage manipulations. The graph plots the behavior of the system under attack computing a set of 10,000 multiplications with randomly selected input operands at different supply voltages. The number of faults increases exponentially as the voltage drops. The graph also reports the percentage of erroneous products that manifest only a single-bit flip.

As expected, the number of faults grows exponentially with decreasing voltage. In the graph of Figure 6 we also plotted the fraction of FWE erroneous computations that incurred only a single-bit fault, as it is required to extract private key information effectively. Note that, with decreasing voltage, eventually the fraction of single fault events begins to decrease as the FWE algorithm experiences multiple faults more frequently. The ideal voltage is the one at which the rate of single bit fault injections is maximized, 1.25V for our experiment. The error rate introduced at that voltage is consistent with the computational characteristics of FWE, which requires 1,261 multiplications to compute the modular exponentiation of a 1,024-bit key. Thus, the attacker should target a multiplication fault rate of about 1 in 1,261 multiplications (0.079%). Using this particular voltage during the signature routine we found that 88% of all FWE invocations led to a corrupt signature.

6.2 Faulty signature collection

In our experiments, we gathered 10,000 digital signatures computed using a 1024-bit private RSA key. Once collected, signatures were first tested to check if they were faulty (by verifying them with the victim machine's public key). Once a faulty signature was identified, it was sent to a distributed analysis framework that implemented the algorithm outlined in Section 5.1. By setting the supply voltage at 1.25V, we found that 8,800 of the 10,000 signatures were incorrect. Within this set, only 12% (1,015 in total) had incurred a single-bit fault in the result of only one multiplication during the computation of the FWE algorithm, leading to useful corrupted signatures for our private key recovery routine. The subset of corrupted signatures that conforms to our fault model is not known a priori, thus all the 8,800 collected signatures had to be analyzed with our algorithm.

The analysis was run on a 81-machine cluster of 2.4 GHz Intel Pentium4-based systems, running Linux. The distributed algorithm was implemented using the OpenMPI libraries and followed a classic master-slave computing paradigm, with one machine acting as a master and 80 as slaves. The master distributed approximately 110 messages to each slave for checking. Individual slaves could check a message against a single potential window value and all fault locations and squaring iterations in about 2.5 seconds. During the analysis, the master directed all slaves to check their own messages for a particular single-bit fault in a particular window of the FWE computation. To reduce the time for synchronizing slaves, we divided their messages into 4 equal-size groups, and processed these groups serially until the value of the key window was found.

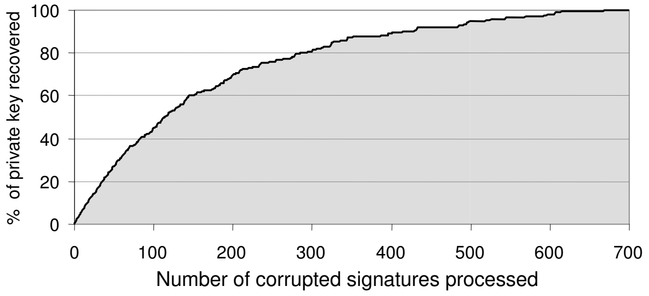

Figure 7: Cumulative percentage of private key bits recovered. To recover the private key in the shortest amount of time, we need to collect at least one corrupted signature for each of the exponent windows. The graph shows the percent of key bits recovered as a function of the number of faulty signatures analyzed.

Figure 7 shows the percentage of the total private key bits recovered, as a function of single-bit faulty signatures processed. As shown in the graph, the full key is recovered after about 650 single-bit faulty signatures are processed. Figure 8 shows the number of single-bit corrupted signatures available for each bit position within the 1024-bit FEW multiplication. We found that the bit errors were skewed towards the most-significant bits of the processor's 32-bit datapath (due to the longer circuit paths used to compute these bits), thus by searching for bit errors in these bit positions first, we could significantly speed up the search process. With our distributed analysis system, our computer cluster was able to recover the private key of the attacked system in 104 hours, for a total of about one year of CPU time. We expect the overall performance of the distributed application to scale linearly with the number of workers in the cluster.

7. CONCLUSIONS

In this work we described an end-to-end attack to a RSA authentication scheme on a complete FPGA-based SPARC computer system. We theorized and implemented a novel fault-based attack to the fixed-window exponentiation algorithm and applied it to the well known and widely used OpenSSL libraries. In doing so we discovered and exposed a major vulnerability to fault-based attacks in a current version of the libraries and demonstrated how this attack can be perpetrated even with limited computational resources.

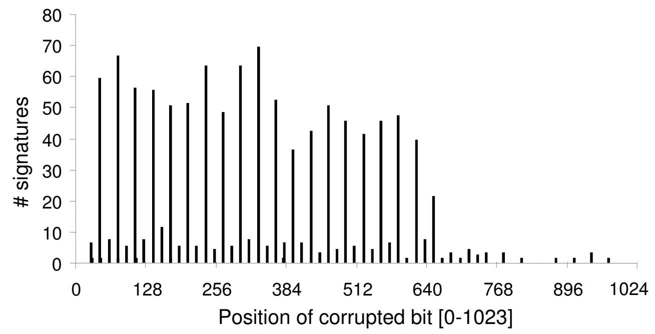

Figure 8: Single bit fault locations in the corrupted signatures. Due to the implementation of the OpenSSL functions and the multiplier used in the processor, the number of locations that might be corrupted in our experiment was limited to only a few locations. This significantly reduced the computational time needed to recover the key, since only a few fault locations have to be tested before the correct result is recovered.

To demonstrate the effectiveness of our attack, we subjected a SPARC Linux system to a fault injection campaign, implemented through simple voltage manipulation. The system attacked was running an unmodified version of the OpenSSL library. Using our attack technique, we were able to successfully extract the server's 1024-bit RSA private key in 104 hours. The work presented in this paper further underscores the potential danger that systems face due to fault-based attacks and exposes a severe weakness to fault-based attacks in the OpenSSL libraries.

Acknowledgments

The authors acknowledge the support of the National Science Foundation and the Gigascale Systems Research Center.

8. REFERENCES

[1] OpenSSL: The Open Source toolkit for SSL/TLS. http://www.openssl.org.

[2] C. Aumüller, P. Bier, W. Fischer, P. Hofreiter, and J.-P. Seifert. Fault attacks on RSA with CRT: Concrete results and practical countermeasures. In *Proc. of the Workshop on Cryptographic Hardware and Embedded Systems*, Aug 2003.

[3] F. Bao, R. Deng, Y. Han, A. Jeng, D. Narasimhalu, and T.-H. Ngair. Breaking public key cryptosystems on tamper resistant devices in the presence of transient faults. In *Proc. of the Workshop on Security Protocols*, Apr 1998.

[4] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer's apprentice guide to fault attacks. *Proc. of the IEEE*, Feb 2006.

[5] E. Biham, Y. Carmeli, and A. Shamir. Bug Attacks. In *Proc. of Advances in Cryptology*, Aug 2008.

[6] D. Boneh, R. Demillo, and R. Lipton. On the importance of eliminating errors in cryptographic computations. *Journal of Cryptology*, Dec 2001.

[7] M. Boreale. Attacking right-to-left modular exponentiation with timely random faults. In *Proc. of the Workshop of Fault Diagnosis and Tolerance in Cryptography*, Oct 2006.

[8] D. Brumley and D. Boneh. Remote timing attacks are practical. In *Proc. of USENIX Security Symposium*, Jun 2003.

[9] K. Hamaguchi, A. Morita, and S. Yajima. Efficient construction of binary moment diagrams for verifying arithmetic circuits. In *Proc. of the International Conference on Computer-Aided Design*, Nov 1995.

[10] M. Joye, A. Lenstra, and J.-J. Quisquater. Chinese remaindering based cryptosystems in the presence of faults. *Journal of Cryptology*, Dec 1999.

[11] A. Menezes, P. V. Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Oct. 1996.

[12] J. Rabaey, A. Chandrakasan, and B. Nikolic. *Digital Integrated Circuits*. Prentice Hall, 2 edition, Jan 2003.

[13] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, Feb 1978.

[14] J. Schmidt and C. Herbst. A practical fault attack on square and multiply. In *Proc. of the Workshop of Fault Diagnosis and Tolerance in Cryptography*, Aug 2008.

[15] D. Wagner. Cryptanalysis of a provably secure CRT-RSA algorithm. In *Proc. of the Conference on Computer and communications security*, Oct 2004.

Appendix A - Proof of Theorem 5.1

From here on, all expressions are implicitly assumed to be $\mathrm{mod} n$, we omit the notation for reasons of space. Define k as the ratio between the number of bits in the private key d and the number of bits w in the window size: $k = \#\mathrm{bits}(d)/w$. The proof proceeds by induction. For the base case, we show that the value of the private key in the most significant window, indexed $k-1$, can be recovered. For the inductive step, we show that, if the value of the private key for windows $i+1$ to $k-1$ is known, then we can recover the value for window i.

Base case. We consider one of the $< m, \hat{s} >$ pairs and we assume that the fault in the corrupted signature $\hat{s}$ was injected during the pth squaring iteration, with $1 \le p \le w$. Hence, from Eq. (3), $\hat{s}$ will have the form:

$$\hat{s} = (m^{d_{k-1}2^p} \pm 2^f)^{2^{w(k-1)-p}} \prod_{j=0}^{k-2} m^{d_j 2^{jw}} \tag{4}$$

The value of d_{k-1} is bound by: $0 \le d_{k-1} < 2^w$. The fault location f can assume any value in $0 \le f < \#\mathrm{bits}(d)$. Finally the squaring iteration p satisfies $0 \le p < w$. Assume that the correct values for d_{k-1}, f and p were known to be d^*_{k-1}, f^* and p^* (the correct values for d_i, $0 \le i \le k-2$ are not known). Then we can multiply both sides of Eq. (4) by $m^{d^*_{k-1}2^{w(k-1)}}$ and obtain:

$$\hat{s} \cdot m^{d^*_{k-1}2^{w(k-1)}} = (m^{d^*_{k-1}2^{p^*}} \pm 2^{f^*})^{2^{w(k-1)-p^*}} \cdot m^d \tag{5}$$

If we raise both sides to the known public exponent e, we obtain:

$$(\hat{s} \cdot m^{(d^*_{k-1})2^{w(k-1)}})^e = (m^{d^*_{k-1}2^{p^*}} \pm 2^{f^*})^{e2^{(w(k-1)-p^*)}} m^{de} \tag{6}$$

$$\hat{s}^e \cdot m^{e(d^*_{k-1})2^{w(k-1)}} = (m^{d^*_{k-1}2^{p^*}} \pm 2^{f^*})^{e2^{(w(k-1)-p^*)}} m \tag{7}$$

It is now possible to search for all triplets $< d^*_{k-1}, f^*, p^* >$ that satisfy Eq. (7), by varying each value within the legal range specified above and checking if the identity holds. Three situations may arise:

1. *No solution is found.* It is possible that no triplet $< d^*_{k-1}, f^*, p^* >$ exists that satisfies the equation. In this case, the pair $< m, \hat{s} >$ is discarded and another one is considered. This situation may arise, for instance, if the corrupted signature $\hat{s}$ was subjected to a fault during an iteration outside the analyzed window.
2. *Exactly one solution.* If only one set of values for d^*_{k-1}, f^* and p^* satisfies Eq. (7), then the value of the private key in the $(k-1)$th window has been found.
3. *More than one solution.* In this case, one of the triplets include the correct d^*_{k-1} value, while the others correspond to other set of values that still satisfy Eq. (7), but do not correspond to the correct private key d on the server side. In this case, the pair $< m, \hat{s} >$ should also be discarded.

Inductive step. The value of the private key d for windows indexed $i+1$ to $k-1$ is known. We want to find the value d_i. We proceed similarly to the base step. From Eq. (3), $\hat{s}$ will now have the form:

$$\hat{s} = \left((\prod_{j=i+1}^{k-1} m^{d_j 2^{(j-i)w}}) m^{d_i 2^p} \pm 2^f \right)^{2^{iw-p}} \prod_{j=0}^{i-1} m^{d_j 2^{jw}} \tag{8}$$

We want to identify a triplet $< d^*_i, f^*, p^* >$ for which d^*_i is the value we are searching for. The ranges for the three values are $0 \le d_i < 2^w$, $0 \le f < \#\mathrm{bits}(d)$ and $0 \le p < k$. To this end, we first assume that we have found such triplet and we multiply Eq. (8) by $\prod_{j=i}^{k-1} m^{d_j 2^{jw}}$:

$$\hat{s} \cdot \prod_{j=i}^{k-1} m^{d_j 2^{jw}} = m^d \left((\prod_{j=i+1}^{k-1} m^{d_j 2^{(j-i)w}}) m^{d^*_i 2^{p^*}} \pm 2^{f^*} \right)^{2^{iw-p^*}} \tag{9}$$

and then raise it to the exponent e to obtain:

$$\hat{s}^e \prod_{j=i}^{k-1} m^{ed_j 2^{jw}} = m \left((\prod_{j=i+1}^{k-1} m^{d_j 2^{(j-i)w}}) m^{d^*_i 2^{p^*}} \pm 2^{f^*} \right)^{e2^{iw-p^*}} \tag{10}$$

Note that all values d_j for $i \le j < k$ are known. There are again three possible outcomes in the search for a triplet satisfying Eq. (10): we only accept $< m, \hat{s} >$ pairs that lead to one and only one satisfying solution.

In conclusion, given a sufficient number of $< m, \hat{s} >$ pairs, it is always possible to find a subset of cardinality k that allows to determine all d_i for $0 \le i < k$. By concatenating the d_i, we obtain the private key d. $\square$

In practice, the situation where more than one solution to Eq. (7) or Eq. (10) is found has extremely low probability and never occurred in our experiments. Complexity and success probability of our attack can be inferred from [6], which targets a different exponentiation algorithm but proposes a similar attack.