Torturing OpenSSL

Valeria Bertacco
{*valeria*}@*umich.edu*

in collaboration with Andrea Pellegrini, Todd Austin,
Armin Alaghi, William Arthur and Prateek Tandon
University of Michigan

Abstract

This document describes a complete end-to-end attack to the RSA signature algorithm on a microprocessor system and demonstrates how input voltage and, in general, hardware vulnerabilities can be exploited to target secure systems. Our system under attack is a SPARC Linux system implemented on FPGA. To perpetrate the attack, we inject transient faults in the target machine by either regulating the voltage supply or temperature of the system. In the attack, we collect a series of corrupted RSA-signed messages that the server generates while being attacked. We then elaborate the data collected and extract the RSA private key. In our experimental evaluation we could extract the system's 1024-bit RSA private key in approximately 100 hours using an 81-machine cluster of 2.4 GHz Intel Pentium4-based systems. This is the first reported physical demonstration of a fault-based security attack of a complete microprocessor system running unmodified production software.

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 [1], 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 or high temperatures, 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

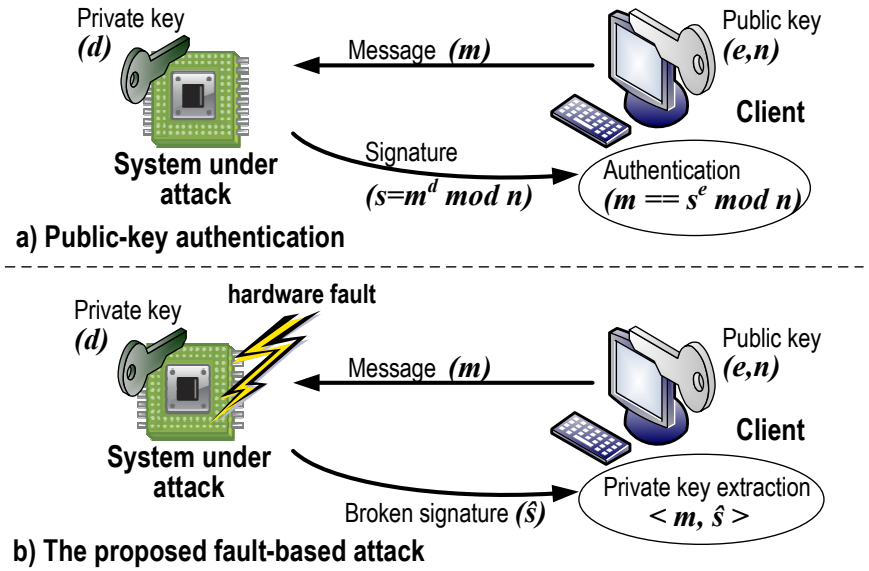

Figure 1: **Overview of public key authentication and our fault-based attack.** a) in public key authentication, a client sends a unique message m to a server, which signs it with its private key d. Upon receiving the digital signature s, the client can authenticate the identity of the server using the 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 $\hat{s}$.

environmental 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. Specifically, we inject faults by either manipulating the voltage supply or the environmental temperature, resulting in occasional transient faults in the 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). When a sufficient number of corrupted messages have been collected, the private key can be extracted through offline analysis. 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.

This project was developed in our reliability research lab at the University of Michigan. The attack first appeared in [8]. Moreover, the study on the impact of temperature variations in generating transient failures was conducted as part of a course project led by Prof. Todd Austin in the Fall of 2010.

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 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).

2 Authentication with RSA

RSA is a commonly adopted public key cryptography algorithm [10]. 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: $m \equiv s^e \bmod n$.

2.1 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 [6]. In this paper we focus on the fixed window exponentiation (FWE) algorithm ([6] - 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 [2].

The fixed-window modular exponentiation algorithm is very similar to square-and-multiply [11], 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.

3 Path Criticality of the Multiplier

In any digital systems there is a particular circuit that is slower than all the others. This circuit can be particularly slow due to the amount of logic gates that the input signals have to traverse to reach the outputs of the circuit, or due to the size of the transistors that compose its logic gates. The time required for an electric signal to traverse the slowest circuit is called "critical path delay" and establishes the upper bound for the maximum frequency of the entire digital system. It is important to notice that the critical path of a component is not stresses all the time that the component is used. Only particular combinations of the inputs will exercise the critical path. For example, the critical path of a ripple carry adder is through the logic that propagates the carry from the least significant bit to the most significant bit. Only when the inputs are such that the carry has to propagate from the first bit to the last, the critical path is stressed. In all the other cases the adder will deliver the correct result in a shorter amount of time. If, due environmental conditions, the timing requirements tighten, the module that holds the critical path is the first one to fail. This can be caused by increasing the clock frequency of the system or by reducing the supply voltage. Reducing the voltage has the effect of making the transistors weaker in driving the output of the logical gates in the design, thus increasing the time necessary for the electrical signals to traverse the logical network and reach the outputs. An elevated temperature has a similar effect: by increasing the resistance of the circuit's components and wires it slows down signals' propagation.

The multiplier unit is one of the biggest and most complex modules in the integer data-path of a simple microprocessor. The VLSI community put a lot of effort in developing high-performance multiplier architectures. Due to the large latency inherent in computing a multiplication, several schemes have been devised to minimize the delay. This effort allowed designers to embed single cycle multipliers in simple but aggressively clocked digital systems such as the microprocessors that we can find in embedded devices and portable phones. Despite the effort of digital designers, in many of these microprocessors the multiplier is still the slowest component in the device, thus becoming the critical path of the whole system.

In this work we exploit this common characteristic of the microprocessor to deterministically cause its multiplier to fail and leak confidential information. It is important to note that several factors might cause a path to be critical over other circuits in the system. Even if the natural critical path of the system is located outside the multiplier an attacker can force the multiplier to slow down such that it holds the critical path. This can be done either by any malicious entity from design time through manufacturing up to end-use.

Designers of cryptographic systems are aware of the danger of alteration of the environment around the secure device. High-end security devices continuously check for changes in temperature, voltage supply and frequency to detect if a physical attack is being perpetrated [3]. However, implementing these protections is very expensive and their cost is justified only for mission critical security devices, such as Trusted Platform Modules (TPM).

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 [9]. 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 7.1 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 2.1). 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):

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

or, equivalently,

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

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 [8]. 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.

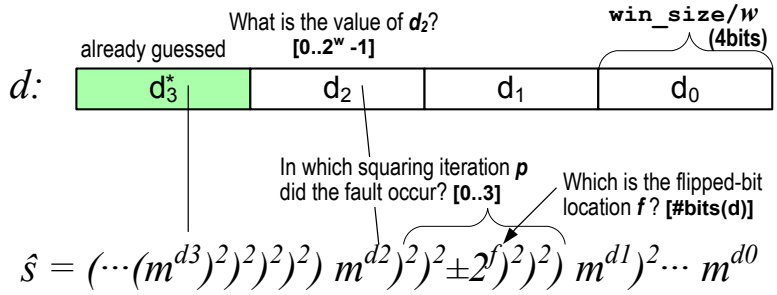

Figure 3: **Example of our private key recovery.** The schematic shows a situation where the private key d to be recovered has size 16 bits, and each window is 4 bits long. Key recovery proceeds by determining first the 4 most significant bits in d, d_3. Then in attempting to recover d_2, all possible 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 the equation below, as derived in the proof in [8].

$$\hat{s}^e \prod_{j=i}^{k-1} m^{ed_j 2^{jw}} = m \left(\left(\prod_{j=i+1}^{k-1} m^{d_j 2^{(j-i)w}} \right) m^{d_i^* 2^{p^*}} \pm 2^{f^*} \right)^{e2^{iw-p^*}} \tag{4}$$

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. (4). 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. (4), 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.

5.2 Computational complexity

We evaluate now the complexity of the private key recovery algorithm. Let us call N the size in bits of the private key d, that is, $N = \#\text{bits}(d)$. Recall that a key aspect of the FWE algorithm used in the encryption process was its constant-time execution, once the value N is fixed. The complexity of the algorithm can then be determined as follow: at the core of the `window_search()` function, is the evaluation of Eq. (4), which requires three modular exponentiations (one for the left side of the equation, and two for the right side, one using the + sign, and one using the - sign). The complexity of this operation can be evaluated by considering FWE, outlined in Figure 2. This algorithm performs $(N+w)$ constant-time multiplications, N in line 6, and additional w in line 9. Thus, the evaluation of Eq. (4) requires $3(N + w)$ constant-time operations. Since in practical situations $N \gg w$, we can safely approximate this to $3N$. In turns, the `window_search()` function performs this evaluation $(2^w \cdot w \cdot N)$ times. Finally `window_search()` is invoked, in the worst-case, $(N/w \cdot S)$ times, where S represent the number of $< m, \hat{s} >$ pairs collected and n/w is the number of windows of the private key. By collecting all these factors together, the complexity of the private key recovery algorithm is, $\mathcal{O}(2^w N^3 S)$. Note that the recovery algorithm has only polynomial complexity on the length of the private key, N.

5.3 How many corrupted ciphertexts?

In the previous section we determined that the complexity of the recovery algorithm is directly proportional to the number of $< m, \hat{s} >$ pairs collected, which we called S. As discussed in Section 5.1, the larger S, the higher the confidence that the recovery will be successful, that is, that there is at least one ciphertext $\hat{s}$ that will reveal the private key window bits d_i for each window i. To compute the number of ciphertexts $\hat{s}$ necessary to cover all the windows with at least probability pr, we use the following reasoning: the probability that window i is not covered by any ciphertexts $\hat{s}$ with a fault occurring in that window, after S ciphertexts have been collected is $(1 - 1/k)^S$, where k is the number of windows in the private key d, that is, $k = N/w$. By using the union bound ($(1 + x) < e^x$ for all x), it follows that the probability that at least one window is not covered after S ciphertexts have been collected is

$$pr = k \cdot (1 - 1/k)^S < ke^{-S/k}$$

Thus, by setting pr to the desired confidence that all windows in the private key are covered, we can solve for the required number of ciphertexts S. For instance, for $pr = 1/2$, at least $S = k \cdot ln(2k)$ ciphertexts are necessary.

Finally, note also that some of the ciphertexts may be unusable because they lead to situations where multiple distinct triplets satisfy the condition of Eq. (4). However, for typical private key sizes, these situations are highly improbable, as discussed also in [1].

6 The Platform Under Attack

To perform our attack we need to be able to inject faults in some of the results output by the multiplier in the attacked microprocessor. The multiplier is a large module within the microprocessor design, often the largest in the integer datapath of a modern processor. Computing a multiplication requires a lot of logic components and in simple CPUs not rarely the multiplier is on the critical path of the system. The component on the critical path of a digital design is the one that requires the longest time to compute the result when its inputs change, thus determining the highest frequency the design can operate. To work properly, a digital circuit should operate at the nominal frequency and voltage. This guarantees that all the signals in the system will have enough time to go through the logic and reach the registers were the value will be stored without missing the deadline imposed by the clock frequency. Raising the frequency or lowering the supply voltage of the system might cause some circuits to miss the deadline clock deadline. In the former case the clock frequency is too high and some electric signals do not have time to go through the whole logic network; in the latter case the strength of the transistors to drive the signals in the logic circuit is not enough to deliver the output on time for the clock deadline. The slowest logic path, and the first to fail in either these cases is the critical path.

Figure 6: **Experimental platform.** A voltage regulator is used to inject faults in a board equipped with a Virtex2Pro FPGA device. The SPARC system is mapped on the FPGA, and it is connected to the server holding a remote file system via ethernet.

6.1 Hardware configuration

The device under attack is a complete system mapped on a field programmable gate array (FPGA) device. An FPGA is a silicon device that can be configured after manufacturing. Beside digital design prototyping, FPGAs are often used as primary support for commercial and military systems, enabling designers to deliver

high-performance upgradable systems without incurring heavy fabrication costs. This system that we target is a complete, unmodified, SPARC-based Leon3 SoC from Gaisler Research, which is representative of an off-the-shelf commercial embedded device. In our experiments, the unmodified VHDL of the Leon3 was mapped on a Xilinx Virtex2Pro FPGA. The board hosts 256MB of DRAM that are used as main memory for the system. Nominally, the Leon SPARC CPU runs at 40 MHz at 1.5 V. A snapshot of our experimental platform is shown in Figure 6.

6.2 Software configuration

Because our prototype lacks a disk interface, we configured our SPARC Linux system as a diskless node. Upon startup the SPARC-processor accesses the kernel from on-board RAM, and after the initial boot sequence, the system mounts an NFS disk available on the network. The NFS disk contains the root partition of the logical file system, which includes a fully functional Debian Linux (Kernel version 2.6.21) installation for SPARC. Among the several applications available, we installed the OpenSSL libraries (version 0.9.8i).

To implement our RSA authentication attack, we make repeated connection requests to the system through the SSH secure shell protocol, which is implemented with the OpenSSL library. It is important to note that in OpenSSL, the FWE algorithm is called as a backup function. If an error occurs while signing a message with the Chinese Remainder Theorem (CRT) algorithm, the FWE implementation is invoked. The result from the CRT algorithm is verified with the public key to make sure that the signature produced is correct. This check was introduced with the purpose of hardening the algorithm to fault-based attacks. We are able to make the CRT algorithm fail using the same technique that we used to inject faults in the FWE computation, thus forcing the OpenSSL function to fall back to using FWE. Surprisingly, in OpenSSL, the result of the backup FWE algorithm is not verified: we will exploit this newly discovered vulnerability in OpenSSL to perform our fault-based attack. Additionally, the OpenSSL libraries utilizes Montgomery multiplications to perform the squaring function within the FWE routine [7]. Injecting single-bit faults into the optimized Montgomery multiplication will expose private key information in a similar fashion as for the traditional multiplication followed by modulo operation.

6.3 Fault injection

There are several ways to inject transient faults into a digital design. A very simple method to trigger faults is to increase frequency or reduce voltage until the electrical circuit begins to fail. Voltage manipulation, in particular, has been shown to be a very effective method of introducing infrequent transient faults into a circuit [4]. Typically, the faults will manifest in the critical paths of the design, thus, the approach has the downside that it can only inject transients into some circuit paths. Another way to create a similar effect is by raising the temperature on chip.

The critical path on the integer pipeline of the Leon3 processor is through the carry signal of the multiplier. Thus, by manipulating the voltage of the power source of the FPGA device, we are able to cause a single bit in the computation of the multiplication to miss the timing requirement. The higher the difference between the nominal voltage and the applied voltage, the higher the probability that the product of an input configuration will be wrongly computed. Injecting faults in the multiplier (or any datapath element in general) has the added benefit that the processor control is not impacted by the faults, which excludes the risk of crashes of the attacked machine during fault injection. We are currently evaluating alternate approaches to fault injection, for situations in which we wish to inject faults into non-critical paths.

7 The Physical Attack

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. Below we discuss the attack using voltage regulation; in the next

section we outline our progress on attacking the same system by leveraging temperature variations.

7.1 Induced fault rate

As we mentioned in Section 4, voltage regulation is critical to an efficient implementation 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 7 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.

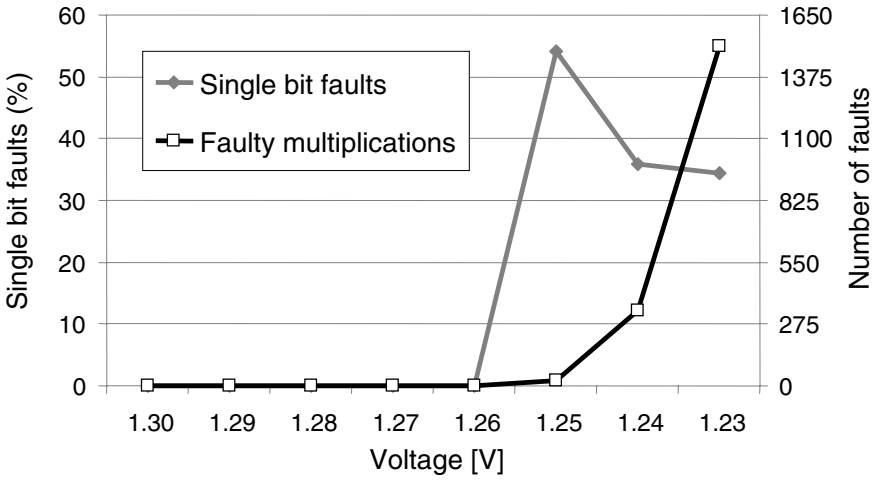

Figure 7: **Sensitivity of multiplications executed in OpenSSL to voltage manipulations.** The graph plots the behavior of the system under attack computing a set of 10,000 multiplications with randomly selected input operands at different supply voltages. The number of faults increases exponentially 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 7 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.

7.2 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.

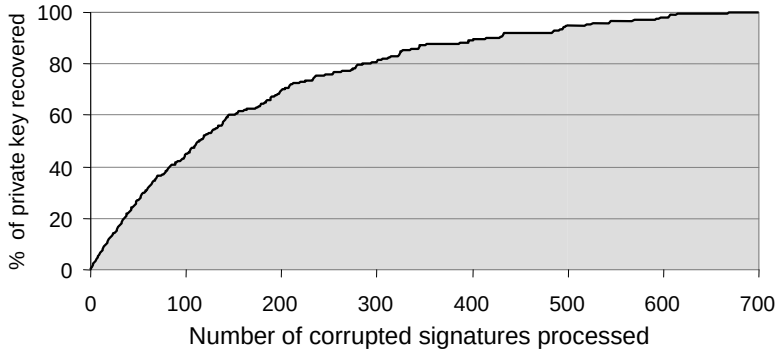

Figure 8: **Cumulative percentage of private key bits recovered.** To recover the private key in the shortest amount of time, we need to collect at least one corrupted signature for each of the exponent windows. The graph shows the percent of key bits recovered as a function of the number of faulty signatures analyzed.

Figure 8 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.

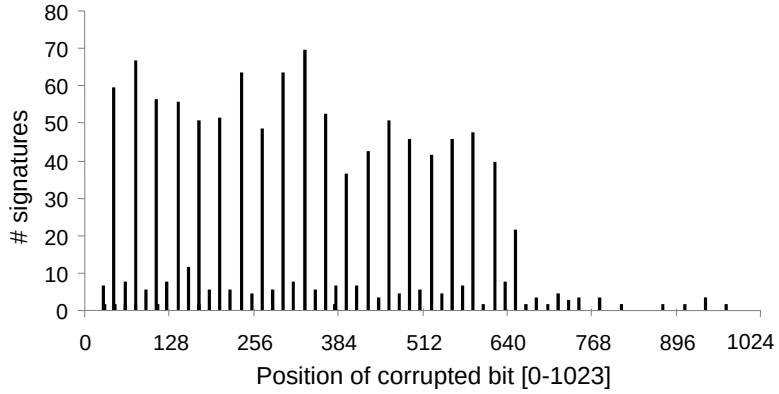

Figure 9: **Single bit fault locations in the corrupted signatures.** Due to the implementation of the OpenSSL functions and the multiplier used in the processor, the number of locations that might be corrupted in our experiment was limited to only a few locations. This significantly reduced the computational time needed to recover the key, since only a few fault locations have to be tested before the correct result is recovered.

Figure 9 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.

8 Attack Via Temperature Control

The attack described in the previous section could be also perpetrated leveraging high temperatures. In this situation signal propagation is slowed down, causing occasional delay faults. Again, since the multiplier has minimal delay margins, it is one of the first units to experience timing errors.

To achieve this goal, we controlled the ambient environmental temperature around the FPGA to cause overheating and errors in computation. We elevated ambient temperature by using two primary sources of heat: an electrical space heater placed in proximity of the FPGA, and an electrical lamp placed directly above the FPGA (the lamp both created and trapped heat). Figure 10 shows this setup. Temperature was regulated by varying the proximity of either device to the FPGA. A multi-meter attached to a temperature probe was used to measure temperature at a 0.1 Celsius resolution. The evaluation of this type of attack was conducted primarily using 128-bit RSA with an initial study of 1024-bit RSA.

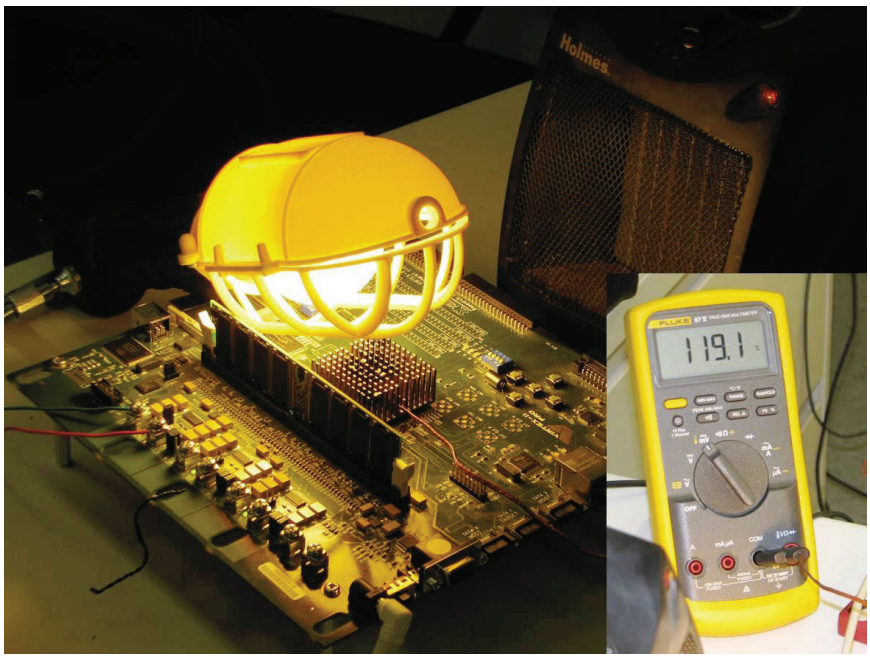

Figure 10: **Controlling temperature on the platform.** A lamp placed directly above the FPGA and a space heater are used to inject faults in this setup. A multimeter with a temperature probe is placed right below the FPGA chip to measure the temperature.

To perpetrate the attack, we would heat of the chip's surroundings, and then allow time for the temperature to equalize throughout the FPGA (>5 minutes). The temperature was closely monitored and regulated by adjusting the position of heat sources. We also found that connecting a heat sink to the FPGA would greatly help the success of this approach, since without it, the FPGA would overheat excessively, providing no useful results for the attack.

In addition, we would also control voltage to lower the temperature necessary to cause the system to fail. However, note that, in this experiment, the voltage was always set so that, if no heat source was applied, the server would sign all RSA messages correctly.

Using a similar setup as described in the previous section, we collected server signatures under a range of temperature and supply voltage conditions, and we gather the number of erroneous signatures. The results of this study are plotted in Figure 11.

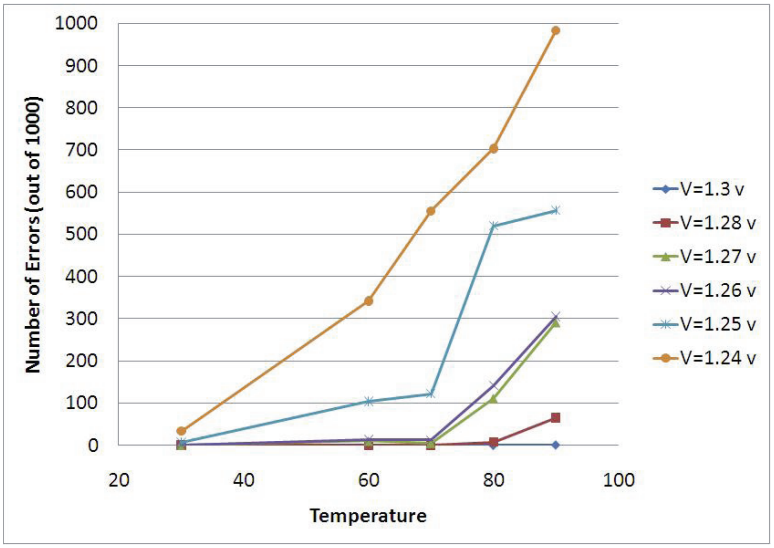

Figure 11: **Error incidence as a function of temperature.** For each operating temperature and voltage, the plot measures the number of erroneous signed messages out of 1,000.

However, as noted in the previous section, our goal is to find an operating point where the system errs only occasionally: indeed, when the error rate is too high, the signed messages we collect tend to include multiple faults and thus not be useful for the purpose of our attack. Figure 12 relates the total number of error messages gathered in each experiment to the fraction of them that it is useful (only one bit flip during only one multiplication) for each operating temperature.

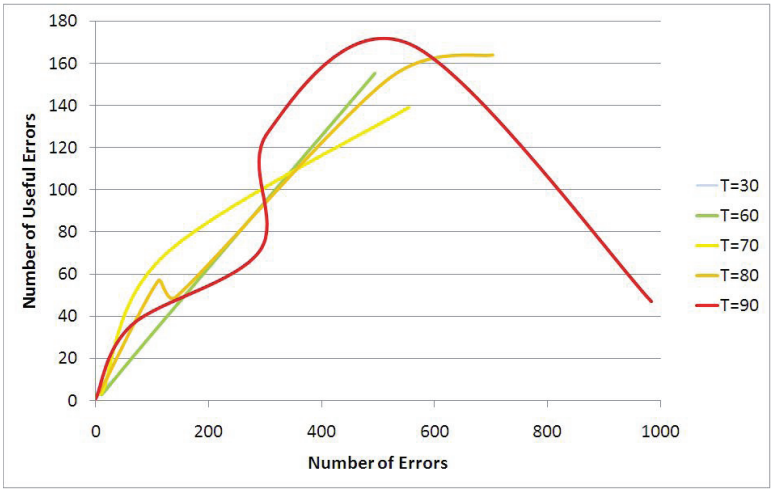

Figure 12: **Number of useful errors vs. number of errors.** As shown in the plot, the number of messages useful for the attack declines when the total number of errors is high – suggesting a wide system failure and multiple bit-flips per signed message.

The overall goal of the experiment was to recover the 128-bit RSA key using a temperature-based attack. Figure 13 shows how many bits could be recover based on the temperature to which the system was subjected.

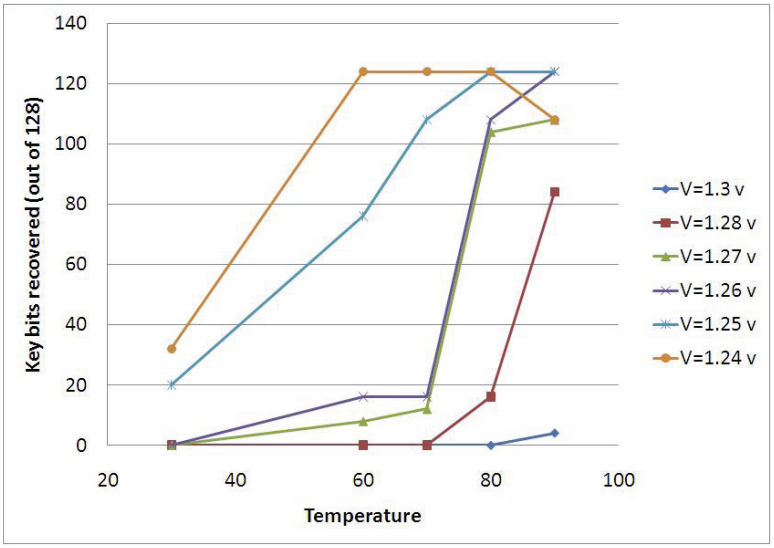

Figure 13: **Secret key bits recovered vs. temperature.** The number of recovered bits of a 128-bit RSA private key is plotted against the temperature on the surface of the processor.

We also explored this attack with 1024-bit RSA. We observed that in this situation the number of errors climbed drastically as a function of temperature, leading approximately step function around 60 degree Celsius. As a results we found it extremely difficult to tune our system to an appropriate temperature for a sufficiently long time to gather all the corrupted signatures we needed to extract the private key. However, we were about to gather 283 corrupted signatures during a 1,000 message requests, 91 of which were useful, leading to the recovery of 30% of the secret key.

9 Countermeasures

Several software techniques exist that could harden the system against the fault-based attack presented in this work. Since our attack requires a malicious user to collect faulty signatures, a system could stop the attack by simply preventing an attacker physical access to the server. While this may work for some servers, other systems cannot be isolated (for instance, publicly accessible systems as the XBox 360 game console). Alternatively, a server could check the integrity of all encryptions before sending out the digital signature. Unfortunately, this approach can be computationally expensive, essentially doubling the computation cost of asymmetric key encryption. Even with these checks, an attacker could still perpetrate a denial-of-service attack on the authentication system by repeatedly injecting faults into the algorithm.

Additional countermeasures have been proposed to prevent attackers from gaining private key data via side-channel attacks. They work by mixing per-request random data (called blinding) into the key at the start of the encryption process, and then removing it after the algorithm completes. Consequently, the secure system does not encrypt the message directly, but rather an alternate value, that can be trivially transformed into the digital signature with an additional operation after encryption completes. The approach noticeably increases the entropy of the RSA computation. Since the attacker does not know the transformations that are performed on the input message, information produced by a corrupted computation will not leak any detail about the private key. The technique was used to harden RSA authentication against timing attacks, and it would also work well for our proposed fault-based attack [5].

10 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 implemented a 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.

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 100 hours. We also studied a similar attack using temperature variations instead of voltage supply. 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.

References

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

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

[3] J. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. Smith, and S. Weingart. Building the IBM 4758 secure coprocessor. *Computer*, (10), 2001.

[4] D. Ernst, N. S. Kim, S. Das, S. Pant, T. Pham, R. Rao, C. Ziesler, D. Blaauw, T. Austin, T. Mudge, and K. Flautner. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proc. of the International Symposium on Microarchitecture*, pages 7–18. Computer, Dec 2003.

[5] P. Kocher. Timing attacks on implementations of Die-Hellman, RSA, DSS, and other systems. In *Proc. of Advances in Cryptology*, pages 104–113, Aug 1996.

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

[7] P. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, pages 519–521, Apr 1985.

[8] A. Pellegrini, V. Bertacco, and T. Austin. Fault-based attack of RSA authentication. In *Proceedings of the Design, Automation and Test in Europe Conference*, Mar. 2010.

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

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

[11] 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.