Software Simulation of Active Attacks on Cryptographic Systems

E.P. Antoniadis, A.G. Voyiatzis
D.N. Serpanos and A. Traganitis

Technical Report
CSD-TR-2001-01

March 11, 2001

Software Simulation of Active Attacks on Cryptographic Systems*

E.P. Antoniadis, A.G. Voyiatzis, D.N. Serpanos† and A. Traganitis
Department of Computer Science
University of Crete

Technical Report: CSD-TR-2001-01

March 11, 2001

Abstract

Cryptographic devices are susceptible to a wide range of attacks, which render them insecure despite their use of strong cryptographic algorithms. Active attacks, which introduce hardware faults (transient or permanent), have been proposed recently and constitute a significant class of attacks. Although they have been analyzed theoretically, they have not been studied either through simulation or in practical implementations.

In this report, we present a simulation study of active attacks. We describe a flexible, scalable software simulation environment, which has been developed for the analysis and evaluation of proposed active attacks as well as for the development of practical methods for their implementation. We present results of the simulation of attacks proposed in the literature, and introduce enhancements and variations that can lead to efficient, practical attacks.

1 Introduction

The successful deployment of a large number of advanced telecommunication and Internet services requires the use of subsystems and devices that enable secure transactions. This is especially important to consumer services, where large numbers of consumers may receive services simultaneously, over public networks. The increasing need for secure transactions has lead to the introduction of hardware devices that perform cryptographic operations, in order

*This work has been partially supported by a Grant from Telcordia Technologies (formerly Bellcore).
†*Contact:* D.N. Serpanos, Dept. of Computer Science, Univ. of Crete, P.O. Box 1470, GR-71110 Heraklion, Crete, Greece. *E-mail:* serpanos@csd.uoc.gr

to establish the identity of the holder, to encrypt sensitive data –typically stored on the device itself–, to implement access authorization, etc. Smartcards constitute a significant category of such hardware devices, and their adoption for general use has led to the development of standards, which allow third party development of technologies and services [10].

The security of hardware devices is based mainly on the use of cryptographic algorithms. Cryptographic algorithms can be divided in two main categories: *secret algorithms* and *public algorithms*. The algorithms of the first category are kept unknown, e.g., the Eurochip algorithms, assuming that the lack of knowledge leads to the infeasibility of their cryptanalysis, and thus guarantees the security of the algorithm. The second category includes algorithms that are publicly known, e.g., DES, RSA, etc. Their security is based on the computational hardness of the underlying mathematical problem that should be solved in order to break the algorithm. It is generally believed that the second category contains more secure algorithms, due to the fact that they have withstood years of cryptanalysis; for example, RSA has been crypt-analyzed for twenty years, without any success [5]. In general, well-known public algorithms, such as DES, RSA, DSS, etc., have been proven secure and have been used in many commercial and military systems. Due to their implementation simplicity, it is feasible to embed them in devices with limited memory and processing power, such as smartcards.

Although public cryptographic algorithms have been proven mathematically secure up to date, their implementation in a system may not lead to a secure system. This is due to "leakage" of information of the physical system, which may compromise the overall security of the system. This was demonstrated first with the introduction of an attack on cryptographic devices –and especially smartcards– based on timing analysis of cryptographic operations [12]. The attack was based on the fact that a processor instruction has variable delay, depending on the values of the operands involved. This variation can leak enough information in order to decide what the instructions' inputs are, and thus recover sensitive information (the secret key or the plaintext message). This work initiated a large number of efforts for cryptanalysis of algorithms executed on hardware devices, and resulted in a new class of attacks, described also by the term *side channel cryptanalysis* [11]. The results of these attacks are quite successful, but up to date they constitute only *certificational weaknesses* of the algorithms, i.e. they are sound weaknesses in theory, but it is very difficult or even impossible to implement them in a real environment.

Side channel attacks can be categorized in two main categories: *passive attacks* and *active attacks*. In the first category, the attacker measures certain characteristics of the device (smartcard), and extracts vital information that renders the system insecure. Such measurements include time variations [12] and power consumption [13]. The difficulty in implementing such attacks is due to the inability to obtain precise measurements associated with specific operations. The problem is especially acute in multipurpose devices, where multiple different operations are implemented. We refer to this problem as the *time isolation problem*, i.e. the

problem of identifying the exact time instant when a specific measurement has to be made.

Active attacks [3] [4] [6] introduce transient faults causing an undetectable change of the intermediate results of a cryptographic operation, which in turn help the attacker to solve the underlying mathematical problem in realistic time. Typically, the underlying mathematical problems are NP-hard and, in theory, the transient faults provide enough information to reduce the complexity of the problem to polynomial time (typically of order less than 3). Although technology to introduce transient faults is available [1] [2], currently, we do not have the ability to introduce such faults exactly in time and space in hardware systems. This means that we still need to solve the *time isolation problem* as well as an analogous *location isolation problem*, i.e. to identify the specific storage element or gate, where the fault should be introduced. Considering these two problems, it seems that such attacks have a higher probability of success in limited environments, in terms of memory and processing power, than in more general, multipurpose devices.

The soundness of some passive attacks has been proven. The attack using measurements of power consumption has been successfully simulated in an appropriate environment [7], and implemented successfully [9] [15]. Additional passive attacks using electro-magnetic radiation measurements, etc. are under investigation [9].

In regard to active attacks, it remains an open question if and under what conditions they can be implemented. In our project we perform a feasibility study of active attacks. Our efforts are directed in two main directions. First, we have developed a flexible, scalable software simulation environment, in order to verify the soundness of the attacks and to develop and evaluate methods for their practical implementation. Second, we have developed an experimental environment where we hold a series of experiments to analyze and evaluate the behavior of a processing environment that executes cryptographic operations under extreme environmental conditions and hazards.

In this report, we describe the results of our first direction, the development of a simulation environment, the verification of the soundness of the transient fault attacks, and the analysis and evaluation of alternatives for their implementation. For our purposes, we have developed a simulator for a smartcard performing cryptographic operations and for a smartcard reader, which introduces hardware faults. We present the results of the evaluation of the effectiveness of the attacks described in [6] and [3]. The metrics we have used include *time* needed to perform a successful attack and *computational complexity* of the process to break the cryptosystem after a successful attack.

The report is organized as follows. Section 2 presents the architecture of the software simulation environment developed for the evaluation. Section 3 describes the implementation of the environment, while Section 4 describes the implementation of the attacks and the results of our evaluations. Section 5 summarizes our conclusions and gives some directions for further work.

2 Software Simulator Architecture

The software simulation environment has been developed to simulate active attacks in cryptosystems using a range of public cryptographic algorithms and to evaluate their efficiency.

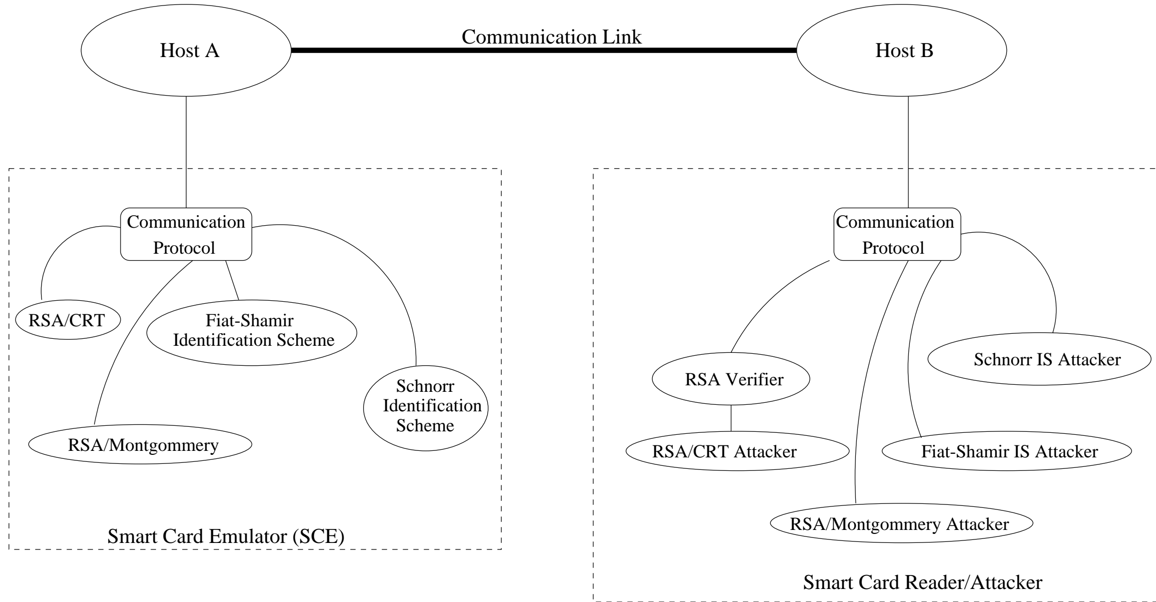

Figure 1: System Architecture

The simulator architecture is depicted in Figure 1. The system contains two components: the SmartCard Emulator (SCE) and the Smartcard Reader/Attacker. Each component (process) consists of a core module, which executes the necessary protocols for interprocess communication, and the cryptographic modules, which perform the appropriate cryptographic operations using the algorithms we analyze.

The SmartCard Emulator emulates a smartcard which may be used for secure transactions, while the Smartcard Reader/Attacker represents an external device, which is used as part of an attack that targets to obtain secret information from the smartcard, specifically to obtain a key used for cryptographic computations. As an example, the Smartcard Reader/Attacker can be a man-in-the-middle subsystem between a smartcard and a service provider, which, if successful in its attack, can obtain services impersonating the smartcard. Clearly, this is not the only way the attack can be exploited in a real system. Furthermore, it should be noted that, a transient fault attack requires two different processes, one that introduces the transient faults in the smartcard and one that behaves as the Smartcard Reader/Attacker. In our system, the SmartCard Emulator introduces the transient faults, while the Reader/Attacker performs the calculations necessary to retrieve the key(s).

There are several transactions that are typically implemented in a complete service provision environment. For the purposes of our analyses, we use two transactions/applications:

the smartcard identification process and authentication. In the identification transaction, we consider that a service provider tries to identify the smartcard. This is typical, because in all systems, services are offered only to "legal" clients (smartcards). In authentication, we consider an application, where the smartcard signs a message for the service provider to authenticate.

The simulation system is implemented on an IBM-compatible PC running Windows 98SE, using Java [20] as the programming language and TCP/IP sockets for interprocess communication. The selection of Java was made for several reasons:

1. it offers great flexibility regarding arithmetic with large numbers, such as the ones involved in cryptographic algorithms;

2. it offers portability, i.e. the ability to execute both processes for the Emulator and the Reader/Attacker on a wide range of platforms;

3. it provides performance comparable to that of C++ for our application, and in some cases, Java is significantly more efficient, when heavy network usage is needed.

4. it offers the Java Card API [22] which will be used in the Java-Card [21], a smart card which will run entirely in Java. It is our intention to extend the experiments on Java-Cards, when they become available.

3 Software Simulator Implementation

The simulator has been designed as a set of two entities, the SmartCard Emulator and the Smartcard Reader/Attacker. The partitioning in these two entities was directed by the abstraction of the attack model: the cryptographic system is one entity, where faults are introduced, while the attacker is a separate entity, which observes the results of the system (faulty or not) and attempts to break the system using that information. In our efforts, we separated these two entities in order to have a flexible platform in which faults can be inserted independently of the observer of the results. Although this approach seems unnecessary in the analysis of this specific report, the system constitutes a flexible environment for experimentation (development and evaluation) of additional attacks.

The flexibility of the system originates also from the modular design of the system's entities. The modular design enables the use of alternative cryptographic algorithms for secure communication between the two entities (e.g., RSA, DES, etc.). This modularity allows the development of additional encryption modules that implement new algorithms of interest. In the following, we describe the modular design of the SCE and the Reader/Attacker.

3.1 SmartCard Emulator (SCE)

The Smartcard Emulator (SCE) is a software component, divided into five Java classes:

- SmartCardEmulator.class
- RSACRT.class
- RSAMontgomery.class
- FiatShamirIS.class
- SchnorrIS.class

SmartCardEmulator is the basic class that acts as the SCE. It supports three cryptographic schemes: the RSA encryption and verification algorithm [16], the Fiat-Shamir identification scheme [8], and the Schnorr identification scheme [18], through use of the appropriate class. RSA is supported for two implementations, one using the Chinese Remainder Theorem (CRT) and one using Montgomery arithmetic.

When SmartCardEmulator is started, it requires 4 arguments: *bit_length*, *certainty*, t and *port_num*. The first two arguments are used for the generation of necessary prime numbers, as described in detail below, while the third argument is the "t" parameter required in the Fiat-Shamir identification scheme, and the fourth argument indicates the port to which the emulator's socket should be bound to. If the emulator is started with inappropriate parameters, then it exits with a typical usage alarm message.

If the command line parameters are correct, then the emulator initializes the four algorithms and binds to a socket at port *port_num* to listen for incoming connections. If any of the algorithms does not initialize properly or the port *port_num* is not available, the SCE exits printing the corresponding error message. Upon successful initialization, the SCE runs an endless loop which accepts a connection, serves it and closes it.

3.1.1 Communication protocol

The system entities (processes) communicate using TCP/IP sockets and a protocol, which implements either the identification or the authentication application mentioned previously.

Upon a successful connection, the client (attacker) transmits to the server two parameters: the algorithm to use for encryption and a parameter indicating whether there should be a transient error introduced. Both parameters are passed as strings, with the second string having the values *true* or *false*. Depending on the requested algorithm, the appropriate exchange of information is implemented. The specific exchange of messages for each algorithm is shown in Figure 2. Every box in the figure represents the transmission of a string. Continued dots (...) represent a series of transmissions of the same type, i.e. in the Fiat-Shamir identification scheme, if $t = 4$, then the transmitted messages are $u_1,\ u_2,\ u_3,\ u_4$.

Figure 2: SCE - client communication protocol

3.1.2 Error generation

The SCE introduces transient faults upon request. A request for a fault in the cryptographic calculations is indicated by setting the second parameter of the SCE communication protocol *true*. The SCE introduces a fault on the appropriate quantity, using the appropriate fault model for the specific experiment and choosing its position with a uniform distribution. The uniform distribution is implemented using the *java.util.random* function of the Java Development Kit [23], with a 48-bit seed (the default constructor uses the current time as seed, but there is a constructor that accepts a user seed, if that is desirable).

3.1.3 Cryptographic operations

We have developed in-house implementations for the four cryptographic algorithms of interest for two main reasons:

- to obtain experience in developing efficient components for these widely used cryptographic algorithms;
- to evaluate the use of Java for cryptographic algorithm implementation. The wide deployment of Java justifies the study of its internal mechanisms for cryptographic operations.

One of the most important and interesting problems in the development of cryptographic systems and the implementation of cryptographic algorithms is the security of the required elements: the random numbers and the prime numbers. In the Java Development Kit 1.2, which we have used, random and prime number generators are built in the *java.math.BigInteger* package. In regard to random numbers, we use the *java.security.SecureRandom* routine, which provides cryptographically strong random numbers by using well-known Pseudo-Random Number Generators (PRGNs). For prime numbers, we use the *BigInteger(bit_length, certainty, random)* constructor, which produces a prime number with *bit_length* bits and which is composite with probability P, where $P < 1/2^{certainty}$. The *bit_length* and *certainty* parameters for prime number construction are the first two parameters in the command line, when starting execution of the SCE. Both of them can be arbitrarily long; longer (larger) *certainty* means greater probability that the constructed number is prime, with the side-effect of a slower running program, as more tests are performed.

3.1.4 RSA with Chinese Remainder Theorem

One of the algorithms that can be executed by the SCE is the RSA algorithm using the Chinese Remainder Theorem (CRT) for exponentiation. The application used is to prove the identity of the SCE in the following fashion. The SCE computes $E = M^s \bmod N$, where M is a known message, and upon successful connection to the Reader/Attacker, it transmits 3

pieces of information: the public exponent e, the public modulo N, and the original message M. So, it is possible for the connected party (Reader/Attacker) to verify the identity of the SCE by comparing M and $E^e \bmod N$.

In RSA with CRT, the modular exponentiation is efficiently implemented using the Chinese Remainder Theorem. The performed calculation is described here for completeness.

Let p, q be two secret prime numbers, s the secret exponent, e the public exponent, and N the public modulo of RSA. Also, let $E_1 = M^{s \bmod (p-1)} \bmod p$ and $E_2 = M^{s \bmod (q-1)} \bmod q$. Then,

$$E = M^s \bmod N = (E_1 q(q^{-1} \bmod p) + E_2 p(p^{-1} \bmod q)) \bmod N$$

So, if one knows $q^{-1} \bmod p$ and $p^{-1} \bmod q$, then one needs to compute four products, a sum, and a modular reduction in order to obtain the desired result. The computation of $q^{-1} \bmod p$ and $p^{-1} \bmod q$ costs $O(\log \log n)$ [19], where n is the bit length of N; however, it can be performed in advance. Thus, the total cost of modular exponentiation using the Chinese Remainder Theorem is $O(1)$.

The code for the transient fault generation is inserted between the computation of E_1 and E_2. If a fault should be generated, then it is introduced in E_1, so that the wrong value of E_1 is used in the computation of E. The fault is a flipping of a single bit.

3.1.5 RSA with Montgomery Arithmetic

```
y = M
z = 1
for (k=0; k < n; k++)
     if ( s[k] == 1 )
          z = z*y mod N
     y = y*y mod N
end-for
return z
```

Figure 3: RSA with Montgomery Arithmetic

This algorithm is the same as RSA with CRT, except that it uses "Montgomery Arithmetic" for the computation of the modular exponentiation. The computation of $E = M^s \bmod N$ is implemented with the algorithm shown in Figure 3. In the algorithm description, n is the bit length of the secret exponent s, and $s[k]$ denotes the k-th bit of s.

The code for fault generation is inserted in the *for-loop* of the algorithm, just before the test of $s[k]$. A fault flips a single bit in s.

3.1.6 Fiat-Shamir Identification Scheme

The Fiat-Shamir Identification Scheme [8] is the most complex from the four implemented schemes. Its security is based on the hardness to compute square roots over Z_N. We describe the scheme briefly (a detailed presentation appears in [8]).

The SCE accepts from the command line parameters the value of t –the 3-rd parameter–[1] and picks two prime numbers p and q. Then, it finds a set of invertible items $s_1, s_2, \ldots, s_t$ mod N, where $N = p * q$. In the system, each s_i is a prime number (not equal to p or q, which ensures that is invertible over Z_N). The public key of the SCE consists of t, N, and the set $U = \{u_i | \ u_i = \ s_i^2 \bmod N, \ i = \ 1, \ldots, \ t\}$.

Whenever SCE has to be identified, it executes the following protocol. First, it picks a random number r and transmits the number $r^2 \bmod N$. Then, it receives a set $S \subset \{1, 2, \ldots, t\}$. Upon reception, SCE computes and transmits $y = r \prod_{i \in S} s_i \bmod N$. The client establishes the identity of the SCE by confirming that $y^2 \bmod N$ is equal to $r^2 \prod_{i \in S} u_i \bmod N$.

Faults are introduced in r, where a fault is a single bit flip. A fault is introduced while waiting for the reception of the subset S. This choice takes advantage of the end-to-end message transmission latency, which allows enough time for an attacker to introduce a fault in a realistic environment. However, an attacker must choose the position of the fault carefully.

3.1.7 Schnorr Identification Scheme

The Schnorr Identification Scheme is very efficient for environments, where the party that proves its identity has limited processing power, while the verifying party has significantly more power. Considering realistic environments, the scheme is well suited for use with smartcards.

The security of the scheme is based on the hardness to compute discrete logarithms over Z_p, where p is prime. Its functionality is based on the challenge-response model. We describe the scheme briefly (a detailed presentation appears in [18]). The SCE finds a prime number p with a known factorization of $\phi(p)$, and a generator g of Z_p^*. Then it picks a random number s and publishes g, p and $y = g^s \bmod p$, as shown in Figure 2. In order to prove its identity, it picks a random number r, it computes $z = g^r \bmod p$ and transmits z to the client (verifier). When challenged by the verifier, SCE receives a challenge number t and responds with $u = \ (r + s * t)$. Then, the client (verifier) can verify the identity of the SCE by computing and comparing $q = (g^r * z^t) \bmod p$ and $v = g^u \bmod p$.

In our experiments, we use $p = 2 * q_1 * q_2 + 1$, where q_1, q_2 are two prime numbers, each with length half the *bit_length*[2]. The generator g is computed with sequential search of the candidates, as described in [17].

[1] In the original paper it is suggested that the two parties agree on t; however, the SCE has the option to agree on a t, and for this reason we choose a single t in our implementation.

[2] p is tested for primality and re-calculated, if it fails the test. The Java test has probability of failure equal to $(\frac{1}{2})^{certainty}$, where *certainty* is the parameter mentioned earlier.

Faults are introduced while waiting for the challenge t by the verifier. Similarly to the previous cases, a fault is a single bit flip in r.

3.2 Reader/Attacker Emulator

The Reader/Attacker Emulator (or, simply Attacker) software component communicates with the SCE through a TCP/IP network connection. It implements the communication protocol shown in Figure 2 and performs the necessary cryptographic operations with any of the supported algorithms. The Attacker is implemented as a Java applet, and contains five classes:

- *BellAttack.class*: the applet interface;
- *Attack_RSA_CRT.class*: implements the computations for the attack on RSA with CRT;
- *Attack_RSA_Montgomery.class*: implements the computations for the attack on RSA with Montgomery;
- *Attack_Fiat_Shamir_IS.class*: implements the computations for the attack on Fiat-Shamir identification scheme;
- *Attack_Schnorr_IS.class*: implements the computations for the attack on Schnorr's identification scheme.

Figure 4 illustrates the Attacker interface[3], where one can choose the host and port to connect to (currently the only available SCE is on host *selini.csd.uoc.gr*, port *6666*). The user can choose one algorithm for cryptographic calculations and whether the SCE should introduce a fault in the computations, through the corresponding check-box. The text-area acts as a logging screen, giving information about the progress of the attack.

4 Attacks

4.1 Attack on RSA with CRT

The attack on RSA with CRT is the simplest to implement. When the user chooses the "error generation" option, the SCE introduces an error, which is detected by the Attacker in the following fashion.

The "correct" message that should be encrypted by the SCE is $M=$ *"100"*. When the Attacker receives the encrypted message E, it computes $M_{rcv} = E^e \bmod N$ and compares M_{rcv} with M. If they are different, then the attack is successful and one of the prime factors of N is $\gcd(M - M_{rcv}, N)$. Due to the location of the introduced fault in the SCE, the output is always p, the first prime factor of N.

[3]The Attacker Java applet is currently available at URL *http://selini.csd.uoc.gr/sce/Attacker.html*.

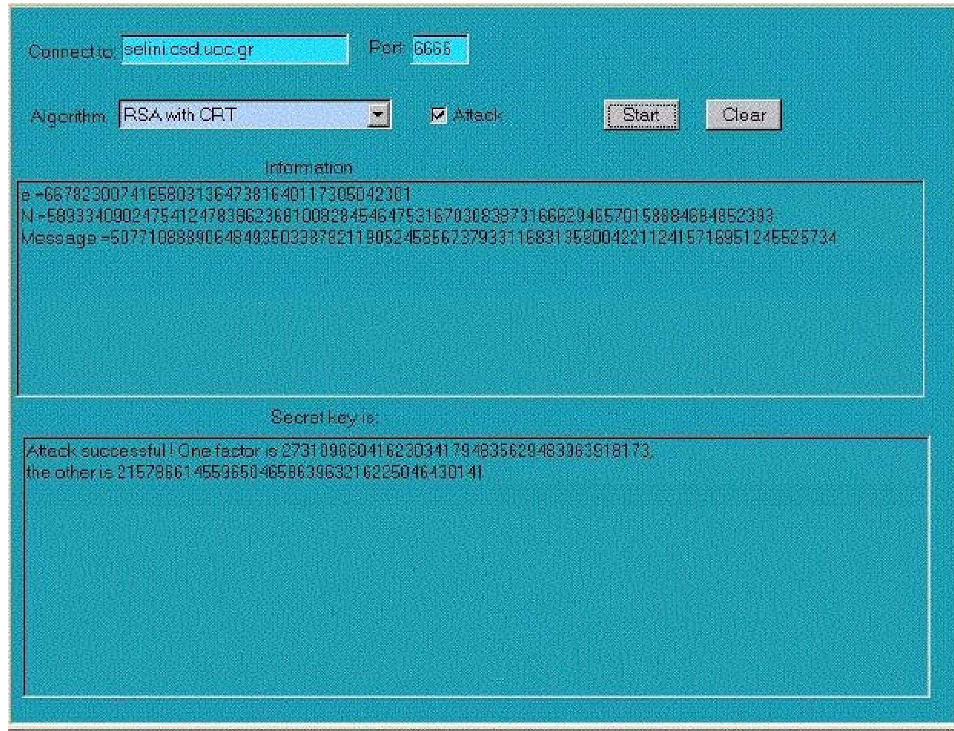

Figure 4: Attacker Interface

This attack is a variation of the attack originally proposed by Boneh et al. [6], in that it breaks the system using only one transaction rather than two. This variation has been proposed by A. Lenstra [14].

The average delay to calculate p and q on our platform is 1.3 sec (including connection setup and protocol execution), when using 64-bit keys. For 512-bit keys, the delay is 1.8 sec on average.

4.2 Attack on RSA with Montgomery Arithmetic

The attack on RSA with Montgomery Arithmetic (RSA-MA) has been implemented using the theory presented by Boneh et al. [6] and Bao et al. [3]. Since the attacks described in the two references are equivalent, we have designed the attack based on the description of Bao et al., because of its conciseness.

Our attack actually differs from the attacks in the literature in one major way. In the proposed attacks, a number of transient faults in introduced (n in Boneh et al. and $O(n)$ in Bao et al.), and the secret key is recovered with a probability of success at least 0.5. In our attack, we introduce a larger number of errors in order to recover the key in full.

The application used in this experiment is authentication: the SCE signs a well-known message with its private key, and the Attacker verifies it by exponentiating the received message

with the public key of the SCE. The attack consists of a sequence of rounds of authentication. In the first round, no transient fault is introduced, so that the Attacker receives the correctly signed message ($csm = m^d \bmod N$). From this message, the Attacker constructs two vectors, $m = (csm, csm^2, csm^4, \ldots, csm^{2^{(n-1)}})$ and $m_inv = \{m[i]^{-1}, \forall i = 0, \ldots, n-1\}$ over Z_N, i.e.

$$m[i] \equiv m^{d^{2^i}} \bmod N, \forall i = 0, \ldots, n-1$$

and

$$m_inv[i] \equiv m[i]^{-1} \bmod N, \forall i = 0, \ldots, n-1$$

In subsequent rounds, transient faults are introduced. Upon reception of a faulty signed message, the Attacker constructs a *ratio*, which is the product of the received message and the inverse of the correctly signed message csm. Then, the *ratio* is compared to $m[i]$ and $m_inv[i]$. If there is an i such that $ratio = m[i]$, then the corresponding bit of the secret exponent is 0. If there is an i such that $ratio = m_inv[i]$, then the corresponding bit of the secret exponent is 1.

In our attack, we introduce a larger number of errors in order to recover the key in full. Given that the secret key has bit-length n and that we introduce faults which are uniformly distributed among the n key bit locations, the expected number of introduced faults is $n \times H_n$, in order to recover all n secret bits.

Clearly, there is a trade-off between the number of introduced faults and the probability of successfully deriving the secret key. In our approach through, we provide improved performance in a realistic system. The reason is that in the proposed attacks, a large number, $O(n^3)$, of RSA encryptions is required to achieve 0.5 probability of success, while in our attack no such encryptions are required. So, in a realistic environment, the processing of the extra faults will be faster than the performance of the RSA computations. However, in our attack there is a requirement for a larger number of introduced transient faults. The success in introducing the increased number of transient faults is certain, because even the introduction of n (or $O(n)$) faults assumes that a mechanism exists that introduces faults reliably in time and in place (i.e., in the secret key); this mechanism can be used effectively for the remaining faults as well.

Key length (n)	Key bits found after $n \log n$ faults	Total number of faults to id complete key	Time per Fault (sec)	Total Time min/max/avg (minutes)
16	15-16	30-85	0.72	0:22/1:00/0:37
64	64	223-373	0.725	3:07/4:37/3:43

Table 1: Fault measurements for RSA-MA key identification

Table 1 summarizes the results of our experiments. The left column in the table represents the bit length of the key, while the second column shows the number of key bits identified after $n \log n$ errors. The third column shows the actual number of faults that were necessary to introduce in order to find the complete key.

The results of this analysis indicate that, one could optimize the attack, so that it does not introduce faults after calculating $(n - k)$ key bits, but use brute force instead. Such kinds of optimizations are desirable, and can be made easily in any realistic system based on its performance parameters.

4.3 Attack on Fiat-Shamir Identification Scheme

We implement the attack proposed by Boneh et al. [6]. We assume that the SCE does not accept singleton sets, S. So, the Attacker must choose the S sets in such a way that the characteristic vectors of the sets form a $t \times t$ full rank matrix over Z_2. After constructing this matrix, the Attacker initiates a series of executions of the protocol using all the characteristic vectors consecutively. For example, if $t = 4$, the transmitted subsets are: $\{1, 4\}$, $\{2, 4\}$, $\{3, 4\}$, and $\{1, 2, 3\}$. After all responses to faults are collected, the Attacker computes the secret set of s_i. The underlying mathematical background and the exact computations performed are the ones described in [6].

Size of N	t	Total Time (sec)	Estimated Attack Time (sec)
512	4	23	10
512	8	37	14
512	16	63	18
512	32	142	40
512	48	225	58

Table 2: Performance measurements for the attack on Fiat-Shamir

Table 2 summarizes the execution time to perform a successful attack on Fiat-Shamir identification scheme on our platform. The reported times correspond to the whole attack process, including the connection and error generation phase as well as the secret parameter computation phase. The estimated attack time is calculated by subtracting from the total time the average delay for connection setup, error generation and secret parameter computation, which have been measured autonomously.

4.4 Attack on Schnorr Identification Scheme

The attack is implemented as proposed by Boneh et al. [6]. For completeness, we decided to not bound the number of errors to $k = n \log 4n$, as proposed in the reference, but perform as many errors as necessary to cover all bit positions of r. This fact changes the complexity of the attack and thus, its running time. For the first phase, which is the collection of responses with errors (due to faults), the complexity is $O(nk)$, where k is the total number of errors. This may give greater complexity than $O(n \log 4n)$, but the probability covering the secret parameter with errors is 1.0 rather than 0.75 of the original attack. For the second phase, the complexity is not affected, i.e. it remains $O(n^2)$ and the probability of success at 0.75. Finally, the time complexity is bounded by the greater of $O(nk)$ or $O(n^2)$ and the probability of success is 0.75.

Length of N	Faults	Attack Time (sec)
16	17-26	51 (0:51 min)
32	71-108	131 (2:11 min)
64	169-276	225 (3:45 min)
128	832	846 (14:06 min)
256	1567	7295 (2:01:35 hours)

Table 3: Fault measurements for the attack on Schorr's scheme

Table 3 summarizes the results of the experiments. The left column shows the bit-length of N, while the right one shows the faults needed to identify the complete secret key.

5 Conclusions

We have presented an efficient and scalable simulation environment for the verification of active attacks on cryptosystems. Using this environment, we verified and evaluated several active attacks, analyzing such parameters as delay, implementation complexity, etc.

Considering the goals of our project, we will continue work in two main directions:

1. extension of the software platform and evaluation of all active attacks that appear in the literature;

2. development of practical methodologies to resolve the problems of *time isolation* and *location isolation.*

In regard to the first direction, we plan to implement the active attacks on the ElGamal and DES algorithms, using the methodologies that appear in [3] and [4], respectively. Furthermore,

we intend to extend our platform to include the JavaCard API using the ISO 7816 communication protocol, in order to identify possible system flaws and to evaluate the efficiency of various attacks on that platform.

The platform gives us the ability to work in an additional direction: to collect statistics in order to evaluate whether there is a relationship between fault positions and the results of encryption. Such statistics may prove useful and enable more efficient or more practical attacks.

References

[1] R. Anderson and M. Kuhn. Tamper Resistance - a Cautionary Note. In *Proceedings of the second USENIX Workshop on Electronic Commerce*, pages 1–11, 1996.

[2] R. Anderson and M. Kuhn. Low Cost Attacks on Tamper Resistance Devices. In *Security Protocol Workshop 97*, 1997.

[3] F. Bao, R. Deng, Y. Han, A.D. Narasimhalu, and T. Ngair. Breaking Public Key Cryptosystems on Tamper Resistant Devices in the Presence of Transient Faults. In *Security Protocol Workshop 97*, 1997.

[4] E. Biham and A. Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In *Advances in Cryptology-Crypto 97 Proceedings*, pages 513–525. Springer-Verlag, 1997.

[5] D. Boneh. Twenty Years of Attack on the RSA Cryptosystem. *Notices of American Mathematical Society*, 46(2):203–213, 1999.

[6] D. Boneh, R.A. DeMillo, and R.J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. In *Advances in Cryptology-EUROCRYPT 97 Proceedings*, pages 37–51. Springer-Verlag, 1997.

[7] J.-F. Dhem, F. Koeune, P.-A. Leroux, Mestré, J.-J. Quisquater, and J.-L. Willems. A Practical Implementation of the Timing Attack. Technical Report CG-1998/1, UCL Crypto Group, DICE, Université Catholique de Louvain, Belgium, 1998.

[8] U. Feige, A. Fiat, and A. Shamir. Zero Knowledge Proofs of Identity. In *Proceedings of 19th annual symposium on theory of computing*, pages 210–217. ACM, 1987.

[9] Cryptography Research Inc. http://www.cryptography.com/dpa/qa/index.html.

[10] ISO. Standards 7816. http://www.iso.ch/cate/d2957.html.

[11] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Side Channel Cryptanalysis of Product Ciphers. Technical report, Counterpane Systems, 1998. available at URL http://www.counterpane.com/.

[12] P. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS and Other Systems. In *Advances in Cryptology-Crypto 96 Proceedings*, pages 104–113. Springer-Verlag, 1996.

[13] P. Kocher, J. Jaffe, and J. Benjamin. Differential Power Analysis. In *Advances in Cryptology-Crypto 99 Proceedings*, pages 388–397. Springer-Verlag, 1999.

[14] A.K. Lenstra. Memo on RSA Signature Generation in the Presense of Faults. Manuscript available from author, arjen.lenstra@citicorp.com.

[15] T.S. Messerges, E.A. Dabbish, and R.H. Sloan. Investigations of Power Analysis Attacks on Smartcards. In *Proceedings of the USENIX Workshop on Smartcard Technology*, pages 151–161, May 1999.

[16] R.L. Rivest, A. Shamir, and L.M. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. In *Communications of the ACM*, volume 21, pages 120–126, 1978.

[17] B. Schneier. *Applied Cryptography*, pages 253–254. John Wiley and Sons, Inc., 1996.

[18] C. Schnorr. Efficient Signature Generation by Smart Cards. In *Journal of Cryptology*, volume 4, pages 161–174, 1991.

[19] D. Stinson. *Cryptography, Theory and Practice*, page 128. CRC Press, 1995.

[20] SUN, Inc. Java Web Site. URL: http://www.javasoft.com/.

[21] SUN, Inc. JavaCard. More information on www.javasoft.com.

[22] SUN, Inc. JavaCard API. More information on www.javasoft.com.

[23] SUN, Inc. JDK 1.2. URL: http://www.javasoft.com/jdk12/api/.