Combined Attacks and Countermeasures

Eric Vetillard and Anthony Ferrari

Trusted Labs, 790 Avenue Maurice Donat, 06250 Mougins, France

Abstract. Logical attacks on smart cards have been used for many years, but their attack potential is hindered by the processes used by issuers to verify the validity of code, in particular bytecode verification. More recently, the idea has emerged to combine logical attacks with a physical attack, in order to evade bytecode verification. We present practical work done recently on this topic, as well as some countermeasures that can be put in place against such attacks, and how they can be evaluated by security laboratories.

1 Forward

This paper presents theoretical work[1] related to the development of attacks that combine logical attacks on the card's software with physical attacks on the card's hardware. This particular piece of work has been performed on Java Card-based smart cards. However, we will see that the new attacks can be applied to other platforms, and Java Card has mostly been chosen because it is the most common interoperable smart card application platform.

2 Background

2.1 Physical Attacks

Smart cards have been subject to physical attacks since their inception. Attack techniques have evolved over time, and new attacks are being invented or enhanced all the time, either on the field (by hackers) or in security laboratories. However, most of the attacks fall in two main categories: observation attacks, whose goal is to observe the behavior of the card through some kind of side channel (timing, power consumption, electromagnetic signals, etc.), and perturbation attacks, whose goals is to modify the behavior of the card, usually by inducing a fault in the silicon.

Around 2000, observation attacks were greatly enhanced by the appearance of complex power analysis attacks, and in particular Differential Power Analysis (DPA, [6]). In the recent years, most developments have focused on fault induction attacks. When these attacks were invented, they were highly unpredictable, with low success rates. Recently, some laboratories have achieved very high success rates on very precise attacks, allowing attackers to design sophisticated attack paths.

[1] This work has been made possible by sponsorship from MasterCard International, and has been practically implemented through a collaboration with the EDSI laboratory, who performed the hardware attacks.

D. Gollmann, J.-L. Lanet, J. Iguchi-Cartigny (Eds.): CARDIS 2010, LNCS 6035, pp. 133–147, 2010.

2.2 Logical Attacks

As smart cards grew more complex, so did their embedded software. In the end of the 1990's, a few systems appeared that allowed multiple applications to run on the same smart card. These systems, such as Multos and Java Card, introduced an additional abstraction layer, a virtual machine, which separated the smart card's system layer from its applications, and allowed the system layer to perform adequate security checks when running applications.

With the ability to load code, these systems also opened the possibility to load malicious code. This led to the development of logical attacks. Although such attacks can be applied on any system that allows code loading, the work has mostly focused on Java Card, for two reasons: first, it has always been an open system, whose specification is readily available; second, Java Card has become the dominant smart card middleware layer.

Commercial security evaluation laboratories, such as Trusted Labs, have used such attacks for several years in the evaluation of the security of smart cards. However, these attacks have remained confidential for a few years, even though the security of smart cards was studied [3] but recent work from the Radboud University Nijmegen [9] and from University of Limoges [7] have made some basic logical attacks publicly available.

There are mostly two categories of logical attacks that can be used on a system like Java Card:

- Illegal bytecode. The attack leverages the fact that Java Card cards are not able to verify that the bytecode that runs is valid (contrarily to other Java virtual machines, running on servers, workstations, or mobile phones). These attacks are quite powerful, and they achieve good results on many platforms.
- Bug exploitation. The attack uses legal bytecode that exploits a bug on the platform. Typical exploitable bugs are buffer overflow bugs, which result from missing checks, usually in lower layers of the card's software. Attack applications of this kind can be verified (their code is correct), and they can therefore be loaded on actual cards. However, they are specific to a given platform.

In both cases, there is a practical limitation today, as very few applications are actually downloaded into smart cards. In addition, the management of applications is restricted on deployed cards, which makes it difficult to load malicious code to perform actual attacks. Attacks based on illegal bytecode face an additional limitation, since application loading procedures usually require the verification of bytecode in order to detect such attacks.

In a typical attack path based on logical attacks, illegal bytecode is used in the attack design phase. Such code is loaded on development cards by the attacker, in order to determine the operational and security characteristics of the platform. At the same time, extensive testing is performed in order to identify an exploitable bug. If both parts are successful, the results are then combined to write an attack application that exploits a platform bug to perform an attack. Of course, since this last attack application exploits a bug, it is perfectly legal, and its bytecode can be successfully verified.

With such attacks, it is possible to cover all typical smart card threats: code and data disclosure, useful for reverse engineering; data modification; and even code modification and injection in some cases, usually successful. Ultimately, in some cases, confidential data disclosure can be achieved.

The illegal bytecode and the exploited bugs have varied effects, but the basic components are often the same:

- Type confusion. This is the most documented attack, which is used to transform integral values into references and vice-versa. The illegal references can then be used to access data illegally.
- Tampering with system data. Modifying an object's descriptor or a stack frame can be used to modify the execution context, and get access to data illegally.
- Illegal jump. Jumping illegally in the bytecode may allow a malicious application to jump into hidden attack code, or even dynamically loaded code, if the jump targets the content of an array.
- Buffer overflow/underflow. A very basic tool, which can be used on arrays, of course, and also on frames and on system buffers.

Of course, card designers have added countermeasures to their implementations, in order to make all of these attacks more difficult. Some of these countermeasures are actually part of the Java Card specification (or companion specifications), whereas some others have been designed by developers in order to make their platforms more robust:

- GlobalPlatform application management. This is a very basic countermeasure, as well as a highly efficient one. Most Java Card platforms also implement GlobalPlatform, which will require at least a cryptographic authentication before to allow any application management operation. This is of course a strong limitation for logical attackers.
- Java Card firewall. A Java Card platform hosts persistent data that belongs to several different applications. The firewall is a mechanism that ensures that an application cannot access another application's data, unless that data has been explicitly shared. The Java Card specification does not mandate any specific implementation, so the strength of the firewall greatly varies among platforms.
- Defensive virtual machines. Making a virtual machine defensive explicitly targets logical attacks, by making the virtual machine perform runtime checks that are redundant with the Java Card bytecode verification. The main issue is here performance, since adding redundancy at the virtual machine level has obvious costs.
- Advanced memory layouts. The first implementations of Java Card mostly used a flat memory model, in which user data was mixed with security attributes, and in which objects from different applications could be mixed. As memory grew and memory management became more complex, the layouts also changed, often becoming a countermeasure, in particular against system data disclosure.

– Classical countermeasures. Naturally, most classical countermeasures are used on Java Card platforms. Confidential objects are stored encrypted, and integrity-sensitive objects are implemented with enough redundancy.

With all these countermeasures, or even with a subset, it is in most cases difficult to identify an attack path based on logical attacks, except on the weakest platforms.

3 Mixing Physical and Logical Attacks

An idea that has been around for a while is to mix physical and logical attacks in order to build a successful attack path on a platform or application. In most cases, the high-level attacks is performed at the logical level, while physical attacks are used to thwart the countermeasures included on the platforms. Here are a few ways in which physical attacks can make logical attacks easier:

– GlobalPlatform attacks. One of the main limitations of logical attacks is that they are based on malicious code. This code cannot be loaded on cards by following standard content management procedures. By attacking GlobalPlatform's cryptographic mechanisms, physical attacks may allow a logical attack application to be loaded on a given platform. Since most platforms don't include on-card bytecode verification, if such attacks succeed on a platform, it becomes possible to include logical attacks in attack paths on such platforms.
– Memory dumps. It is possible to perform partial memory dumps using logical attacks during the reverse engineering phase; however, physical attacks provide ways to perform more complete memory dumps, in particular of the EEPROM.
– Execution trace analysis. It is possible, for instance by using power analysis [10], to figure out the sequence of instructions that is executed. Such attacks allow attackers to know precisely the sequence of bytecode instructions that runs, and to understand where to attack.

Similarly, logical attacks can make physical attacks easier, at least in a laboratory setting. The ability to load a well-known application, designed to be attacked, greatly simplifies the attacker's job. For instance, designing an attack on a counter is much easier on a platform where the attacker has the ability to reset the counter at will.

Overall, an attack scenario is likely to comprise many steps, in which logical and physical attacks will be mixed. In a typical laboratory scenario, we may start by getting a memory dump by a physical attack, then design a few logical attacks in order to get a better understanding of the platform's protections, and finally use a physical attack on a specific part of the code.

Over the years, we have developed a specific methodology for performing evaluations in which logical and physical attacks are mixed, by collaborating with several hardware labs. In a typical evaluation, we provide the code for

the attack applications, as well as the guidelines for performing the attacks (how the attack point can be located, how the attack should be performed, and which results are expected). The hardware laboratory performs the attack (which often requires state-of-the-art techniques), and returns us the results. We then analyze the results obtained (for instance, a memory dump), and then provides the laboratory with new guidelines.

With such collaborations, it has been possible to exploit a wide range of software vulnerabilities, beyond traditional areas such as user authentication and cryptography.

4 Combined Attacks

It has now become possible to push the collaboration between hardware and software laboratories one step further, by designing attacks that can be applied in a systematic way on smart card application platforms. Such attacks have been designed by making the hypothesis that specific hardware attacks are possible, and by then determining the possible effect of these attacks on application middleware. The result of this analysis is a list of possible vulnerabilities, and an evaluation then consists in figuring out which ones are actually available on a given platform.

This approach has always been possible, but two factors have made it more interesting in the past few years:

- Attacks have become more accurate. In particular, the accuracy and success rate of fault induction attacks have greatly improved, making these attacks exploitable.
- Smart card software has become much more complex. Several application platforms are available, and the current shift to flash-based smart cards is going to make application platforms available even on low-end smart cards.

Barbu has applied this technique to Java Card 3.0 [1]. In the present paper, we apply it to classical Java Card 2.2 cards. We first make assumptions that are consistent with today's state-of-the-art on fault induction attacks [5], as follows:

- It is possible to perturb (with a good success rate) a mutable persistent memory read operation in order to replace the read value by a constant 00 value.
- It is possible to perturb (with a good success rate) a specific conditional jump instruction in order to "fix" its result and have it always jump in the same direction.

These assumptions are not unrealistic today, and they are likely to become even more realistic in the future. We will first present a few of the attack scenarios that become possible with such combined attacks.

4.1 Introducing a NOP

The first attack is based on a really simple fact: in Java Card, the opcode nop is represented by the value 00. Since application bytecode is read from mutable persistent memory, our first hypothesis implies that any opcode in the bytecode stream can be replaced by a nop opcode, with two possible consequences:

- If the replaced opcode has no parameters, the instruction it represents is skipped.
- If the replaced opcode has parameters, then its first parameter byte will be interpreted as an opcode, and the following bytes as its parameters.

We therefore see that this simple attack allow us not only to skip instructions, but also to introduce instructions at will in the bytecode stream.

The main interest of this attack is that it targets the main bytecode execution loop. This piece of code is critical for the performance of the platform, and it is therefore very difficult for developers to defend them with complex countermeasures.

4.2 Dropping the Firewall

The second attack focuses on the firewall checks, in order to allow an illegal access on an object through the firewall. In most object-related instructions, the firewall checks consist of a series of checks; the access is allowed if one of them succeeds.

For instance, the invocation of a virtual method is authorized if the object is owned by an applet in the currently active context, or if the object is designated as a Java Card RE Entry Point Object, or if the Java Card RE is the currently active context.

The attack here consists in using our second hypothesis to attack one of these checks, and to make it positive. The method invocation then becomes possible.

This main interest of this attack is that some platforms rely on encryption to protect the applications' assets, rather than on the firewall. In such applications, the firewall tests often include little redundancy, and can therefore be attacked.

4.3 Combining the Attacks

The attacks described above allow us to introduce unverified code in an application, (and in particular to forge a reference), and to "remove" the firewall tests on a reference access. By combining both attacks, and by also including traditional attacks, we are able to build interesting attack paths, such as the following one:

- Using a memory dump, determine the reference of a key object from a sensitive application, or a way to determine the value of that reference. This first part of the attack is performed in a "traditional" way on development cards, or partly on attacked cards, once a dump method has been discovered.

- In an apparently legal application, forge a copy of that reference. We will show in the practical results how this can be achieved by injecting a nop in an apparently normal program.
- Invoke the getKey() method, which returns the key value, on that reference, removing the firewall test. This can also be achieved by a simple attack, this time exploiting our attack on the firewall.
- Output the key value.

This is a very simple scenario, one that we have actually implemented. However, this is just one way of combining these attacks, leading to different attack paths. Our objective is here to show that, provided that perturbation attacks reach a certain quality threshold, an entire range of new attacks becomes available on open cards.

5 Experiments

Experiments have been conducted on a common Java Card platform, with a medium level of security. The targeted implementation is not considered as defensive, and its implementation of the Java Card firewall is rather simple, apparently without any significant redundancy. Quite likely, the typical firewall test consists in retrieving the object context from the object's header, and then comparing it with the current execution context, as required by the specification. This implementation is typical of cards developed by smaller vendors. Such cards can achieve security certifications, at least private ones, usually by relying on cryptography to protect the integrity and confidentiality of sensitive data.

Our experiments on the target platform have shown that it is very easy to identify the firewall test very precisely (it is easily identified as "the" test that fails when there is a firewall error). Breaking this test is therefore a rather simple task for a fault induction specialist, and the hardware laboratory has succeeded several times with good success rates (about 10%, without being detected).

With that result, on our target card, we have the ability to illegally access any object from any application, once we have obtained a reference to that object. The next step therefore consists in building a forged reference to an object. The way in which we can do that is to use another attack to transform an integral value into a reference. We will do that by writing a legal program, which can be transformed into an illegal one through a simple fault induction attack. The attack is based on the fact that the opcode for the nop bytecode is 00, and that we are able to replace a value read from EEPROM by a 00. Let's for instance consider the following code:

```
byte KEY_ARRAY_SIZE = 0x77;

Key getKey(short index){
  if (index<KEY_ARRAY_SIZE) {return keys[index];} else {return null;}}
```

This is a rather inconspicuous sequence of code, which simply performs a range check before to read a value from a local array of references. The bytecode sequence generated for this method is as follows:

```
00 1D     sload_1
01 10 77  bspush 0x77
03 6D 08  if_scmpge +08
05 AD 01  getfield_a_this objects
07 1D     sload_1
08 24     aaload
09 77     areturn
0A 01     aconst_null
0B 77     areturn
```

The attack simply consists in replacing the first bspush opcode by a nop opcode. The executed sequence then becomes:

```
00 1D     sload_1
01 00     nop
02 77     areturn
```

We have achieved our goal of transforming an integral value into a reference. In this particular case, we even have transformed a key index (possibly fetched directly from an incoming command) into a key object.

Once we have forged a reference, we simply need to invoke a method on it (for instance, the getKey() method), and to "remove" the firewall test.

We have implemented a similar scenario in practice, and we have actually been able to disclose the value of a key owned by another application with a success rate of about 5%. In addition, such an attack can be performed step by step, using different APDU commands. This greatly reduces the constraints on the hardware attack, because every fault induction attack can be performed independently of each other.

6 Protecting against the Attack

6.1 Preventing a Full Attack Path

We have shown and implemented an example of a combined attack that works on a basic implementation of Java Card. However, we are still missing some steps in order to industrialize the attack:

- The attack is able to disclose the value of a key whose reference is already known to the attacker. This implies that the attacker must also have the ability to analyze the target application, for instance by dumping the card's memory or analyzing the execution of code [10].
- The attack includes conspicuous code that would not go through a code review, even a basic one. In our example, the value of a key is sent unencrypted to the outside. This would raise questions from any code reviewer.

This dormant code is in fact equivalent to a Trojan horse, waiting to be activated.

Combined attacks allow a wide range of different attacks, but all the attacks that we have looked at have at least one of the following characteristics:

- They include a test whose result needs to be changed for the attack to succeed. This implies that the code that follows this test cannot be reached (dead code), or at least that it cannot be reached in a given case (dead path).
- They include a piece of code that conceals another one, which usually performs an unusual operation, or at least gives access to an unusual operation.

All these characteristics provide us with two leads to defend efficiently against such combined attacks: defensive virtual machines and application code analysis. We will review these two possibilities.

6.2 Defensive Virtual Machines

The idea behind a defensive virtual machine is that it will be under attack. It therefore includes countermeasures that make it hard to build a successful attack path. They do not defend against a specific and well-identified attack, but they rather reinforce the general robustness of the virtual machine.

- Firewall design. The Java Card specification only mandates a few simple checks to implement the firewall. It is also possible to implement the firewall using additional security mechanisms, such as memory encryption or MMU's.
- Redundant checks in the virtual machine. Bytecode verification makes most runtime checks useless under standard hypotheses. In a defensive virtual machine, the assumptions are weakened, and some checks are reintroduced. For instance, the virtual machine may verify that all jump targets are legal.

The difficulty in designing a defensive virtual machine is not in finding new countermeasures to include in the virtual machine, but rather to find a good compromise between security and performance. Each countermeasure has a cost (in memory, execution time, or both), which reduces the performance level of the card.

For instance, checking that the destination on a jump is in an appropriate range is very costly if performed on all jumps. However, this can be optimized by only checking the "long" jumps, which are used to jump over 128 bytes in one direction or another. Such jumps are far less frequent than short jumps, and they are far more likely to be involved in an attack, because they can reach a large part of memory.

Beyond local optimizations, the most difficult part in designing a defensive virtual machine is to build an impenetrable web of countermeasures, and to ensure that attackers are very likely to encounter one of the measures when designing new attacks. Some of the cards that we have evaluated greatly succeeded at this, making logical and combined attacks very hard to implement on their cards.

On the other hand, we also have evaluated many cards that are vulnerable to logical attacks, and whose sole protection against them is in the robustness of their GlobalPlatform implementation and the encryption of sensitive data. That means that such cards could be vulnerable to combined attacks, since the applications used in these attacks are valid and can be verified and legally loaded on a card.

6.3 Static Analysis

Application code analysis comes in complement to on-card runtime verifications (defensive virtual machines) and others hardware countermeasures, by statically ensuring a code being present on-card is not malicious. Static verification of programs is a very active research field, in particular around Java.

The Java bytecode verifier is the most commonly used static analysis tool. However, it is a very simple analyzer, and it remains possible to prove many more properties on Java bytecode. If it has been demonstrated that static verification could be performed on-card during code loading [8], bytecode verification remains traditionally performed off-card and has to be included in organizational countermeasures (bytecode verification and signature).

We have been working since 2002 on a tool that verifies the portability and security of Java Card applets. This tool has been used in many evaluations for the validation of numerous security rules e.g. tracing the use of critical APIs. Over the years, it has been extended mostly in collaborative research projects with information flow analysis [2] and security contract verification [4].

Static analysis tools for Java Card work by running analysed applications on an abstract virtual machine, and then checking that some rules can be verified. The abstraction can be quite precise, with an execution model that performs a global analysis of the code (in contrast to a method-local analysis), and a precise representation of the heap. This is possible because in Java Card, most of the objects are statically allocated, and the depth of invocations is limited by the small size of the Java stack.

We here suggest using a static analysis tool in order to identify applications that may contain Trojan horses to be used in combination with a physical attack as for instance

- Identify dead code and dead paths.
- Identify code that output secrets, in particular key values.
- Identify "unusual" constants.
- Identify unused variables.

7 Analysis

7.1 Can the Attacks Be Reproduced?

Yes, they can. On the logical part, implementing these attacks only requires a good level of expertise on smart card weaknesses and an in-depth knowledge

about the implementation of Java runtime environments. On the physical part, implementing these attacks simply requires an ability to perform very precise fault induction attacks.

In both cases, few laboratories have the required skills in early 2009, and we don't know any laboratory that has both skills. Nevertheless, these skills are likely to become more common in the future.

7.2 Is Java Card Less Secure Than Other Systems?

In terms of resistance to attacks, there is no reason to believe that the Java Card platform is more sensitive than other platforms to our attacks. With the kind of precision that physical attacks can now achieve, all other existing middleware become vulnerable, as soon as they include the ability to load code.

However, a few factors make Java Card a common target for such attacks:

- Greater availability. Getting a development kit for a Java Card platform is quite easy, and the Java Card specification is also available to everybody.
- The "Windows effect". Just like Microsoft Windows, Java Card is today's dominant platform, so there are more incentives for attackers to develop an attack against Java Card.

This status may evolve with the gradual deployment of Java Card 3 platforms. The new platform includes new countermeasures, like mandatory bytecode verification, but it is also exposed to many new kinds of attacks. In addition, as it is new, it will be easier to find exploitable bugs, at least for a while.

About other platforms, the minimum requirement for being able to perform the attacks described here is to be able to develop an attack application and run it on a development platform, which must be similar enough to the platforms being actually deployed. In addition to traditional "open" Java Card-based smart cards, this covers the following cases:

- Any open card based on downloadable applications, such as BasicCard or Multos.
- Any closed card based on one of these open systems, provided that the application code runs in the same way on these cards.
- Any card OS that can be customized using an open API, as we can expect to find on flash-based smart cards.

Although they are nice-to-have features, the following are not absolute requirements for these new attacks to work:

- The ability to load applications on the attacked cards. Once an attack pattern is known, it is easiest to load an application that contains the pattern on an attacked card. However, it is also possible to analyze a closed card's code in order to find and exploit an instance of the attack pattern.
- A bytecode interpreter. The attack must be applied on a generic abstraction layer, but this layer does not need to be a bytecode interpreter. For instance, our firewall attack could work just as well on any memory management API that provides isolation between applications.

Of course, openness leads to an increase of the risk. However, developers of high-end open cards are likely to be aware of these risks and to include appropriate countermeasures, whereas developers of low-end closed cards are more likely to include very basic security measures, leaving more vulnerabilities available for exploitation.

7.3 Can Platforms Be Protected?

Of course, it is possible to include adequate protection in Java Card platforms. The main consequence of the new combinations of attacks described in the present paper is that some implementations become too vulnerable, and need to be replaced by alternative implementations. For instance, implementing firewall checks by a simple comparison between the current context and an object's owning context becomes almost impossible to protect.

The main challenge for platform developers will be to design new countermeasures that are efficient against combined logical and physical attacks, while providing an adequate performance level. This will in some cases require a significant redesign of the platform, with the introduction of new countermeasures.

However, this kind of situation occurs with all new attacks. For instance, in order to protect against DPA attacks [6], a well-known countermeasure consists in introducing counters that limit the number of uses of any given key. In that particular case, the countermeasure that protects a cryptographic implementation does not strengthen the cryptographic algorithm itself; it just makes the new attack unpractical.

7.4 What Security Measures for Certified Platforms?

In order to certify a platform in the context of this new attack, we need to look for suitable security measures on the platform, and to ensure that the proper assurance measures are taken in order to guarantee that these measures are theoretically suitable and that they have been correctly applied on the platform.

Reverse engineering is a key part of most attack paths, as the attackers need some information about the application and about the platform. Limiting the amount of information leaked from the platform then becomes an objective by itself. Protecting the confidentiality of the application bytecode remains difficult, since we have seen that power analysis allows significant disclosure. On the other hand, power analysis is very time-consuming, and it remains interesting to include protections against the more efficient alternative for reverse engineering: memory dumps. There are at least two interesting leads to get that protection:

- Protecting against classical physical memory dumps. The code targeted by memory dumps usually includes a transfer of memory (for instance, copying an array). These operations are not numerous. They should be carefully protected, and evaluations should verify the efficiency of the countermeasures.
- Protecting against logical memory dumps. Standard logical attacks are often used in the reverse engineering part, because they are very efficient (once an attack works, it is entirely deterministic, and it can be exploited in limited

time). The idea would here be to ensure that development cards, which are typically accessible to hackers, only accept verified bytecode. This can be achieved by embedding a bytecode verifier, or by embedding a mandatory DAP verification, and by having all applications verified and signed, either manually or automatically.

Another potential direction is to make the virtual machine more defensive, in particular regarding firewall checks. The objective is here to limit the potential harm made by an application to its own data. It is difficult to design systematic protections of the bytecode, mostly because of the high cost. For instance, adding redundancy to the main bytecode interpretation loop (for instance, to check the integrity of the opcode) will incur a significant performance cost. There are nevertheless a few leads:

- Protecting the firewall tests. The firewall tests are executed often, but only on object accesses. It is therefore possible to introduce some level of redundancy without paying a very high cost. The laboratories should focus on these tests, and perform fault induction tests on them.
- Protecting the virtual machine locally. Even if global countermeasures are very costly, it is possible to introduce some countermeasures that make the attacks more difficult, for instance performing some checks when running a nop opcode, which is very unusual in normal applications. The laboratories should look for such protections during evaluations.

The examples above are only instances of the checks to be performed. Yet, they show that in most cases, the countermeasures are indirect, and require a good understanding of combined attacks to be designed, implemented, and tested.

7.5 What Security Measures for Certified Applications?

In order to certify an application in the context of this new attack, the objective for a laboratory is to ensure that the application does not include any Trojan horse, and that it does not contain any code that has been designed to be attacked.

It is possible to introduce both intentional weaknesses and attack code in applications, even if it undergoes a security certification. Since such attacks require an insider job from the developer's staff, all countermeasures need to be implemented at the laboratory level. The laboratory therefore needs to consider bytecode-level attacks in their analysis of the application code. There are at least two things to watch:

- Analyze thoroughly unusual or useless code. If a piece of code does not seem to perform any interesting computation, it may be part of a Trojan Horse. Evaluators should be careful about this when performing a code review.
- Consider bytecode-level attacks. In addition to the obvious application weaknesses, which are visible at the application level, evaluators should be careful about bytecode-level attacks, which may either be voluntary or not.

The last point is very important. Although we have focused here on voluntary injection of vulnerable code, these attacks can be applied to any code, and they make it quite easy to modify the outcome of a program. Such fault induction attacks are already possible, but the attacks on bytecode may be easier to implement in some cases, for instance when attacking memory read operations has a good success rate.

7.6 Conclusion and Future Work

Logical attacks on smart cards have been available for many years. However, their practical application has been limited by the fact that, in order to be applied, they required an attack on the GlobalPlatform card content management commands, which are often difficult to implement.

With combined attacks, this work demonstrates that it becomes possible to implement a full attack path by combining logical attacks with a standard fault induction attack. Such combined attacks can therefore be implemented on platforms that were until now considered sufficiently safe.

We have shown a complete attack path based on combined attacks, which we have implemented on a classical platform. The results presented here remain incomplete, in the sense that we have mostly worked on code in which we have planted attack code without making significant efforts to conceal it. Therefore it would be of interest to work on attacks that can be concealed in a standard application, as well as to work on the exploitation of combined attacks on standard applications.

The platform on which we have performed the attacks has received many security certifications, while being known as sensitive to logical attacks. The availability of combined attacks raises the question of the requirements for certifications of open card platforms in the future. Evaluation laboratories should determine how these attacks should be considered in standard security evaluation programs.

Finally, another challenge raised by this paper consists in working on countermeasures against these attacks, and in particular on the detection of vulnerable code, totally or partially automated. Evaluation laboratories will however be confronted to a double problem:

- Certifying sensitive applications. In that case, laboratories have access to the application's code and specification, and significant resources are allocated to the evaluation. In such case, a partially automated analysis tool, identifying potential vulnerabilities is most useful, even if it possibly leads to false alarms.
- Scanning non-sensitive applications. In that case, laboratories only have access to the application's binary code, and the resources allocated to the evaluation are limited (a few hours). In such case, an automated tool that only detects obviously malicious code is needed.

Improvements of static analysis tools and application evaluation strategies have to be made to take combined attacks in consideration.

References

1. Barbu, G.: Fault attacks on a java card 3.0 virtual machine
2. Barthe, G., Beringer, L., Crégut, P., Grégoire, B., Hofmann, M., Müller, P., Poll, E., Puebla, G., Stark, I., Vétillard, E.: Mobius: Mobility, ubiquity, security. In: Montanari, U., Sannella, D., Bruni, R. (eds.) TGC 2006. LNCS, vol. 4661, pp. 10–29. Springer, Heidelberg (2007)
3. Chaumette, S., Hatchondo, I., Sauveron, D.: Jcat: An environment for attack and test on java card&trade. In: Proceedings of CCCT 2003 and 9th ISAS 2003, vol. 1, pp. 270–275 (2003)
4. Dragoni, N., Massacci, F., Schaefer, C., Walter, T., Vétillard, E.: A security-by-contracts architecture for pervasive services. In: Proceedings of 3rd Int'l Workshop on Security, Privacy and Trust in Ubiquitous Computing (SecPerU 2007). IEEE Press, Los Alamitos (2007)
5. ITSEC Joint Interpretation Library (JIL). Version 2.7. Application of Attack Potential to Smartcards (February 2009)
6. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
7. Lanet, J.-L., Iguchi-Cartigny, J.: Developing a trojan applet in a smart card. Journal in Computer Virology 6(1) (2009)
8. Leroy, X.: Bytecode verification for java smart card. Software Practice & Experience 32, 319–340 (2002)
9. Mostowski, W., Poll, E.: Malicious code on java card smartcards: Attacks and countermeasures. In: Grimaud, G., Standaert, F.-X. (eds.) CARDIS 2008. LNCS, vol. 5189, pp. 1–16. Springer, Heidelberg (2008)
10. Vermoen, D., Witteman, M.F., Gaydadjiev, G.: Reverse engineering java card applets using power analysis. In: Sauveron, D., Markantonakis, K., Bilas, A., Quisquater, J.-J. (eds.) WISTP 2007. LNCS, vol. 4462, pp. 138–149. Springer, Heidelberg (2007)