Static Analysis of Java Cryptographic Applets

Pierre Boury[1] and Nabil Elkadhi[2]

[1] GIE Dyade,
Domaine de Voluceau - Rocquencourt - B.P. 105
78158 Le Chesnay Cedex France,
e-mail: Pierre.Boury@dyade.fr,
home page: http://www.dyade.fr/fr/actions/VIP/pb_homepage.html
[2] EPITECH,
14-16 rue Voltaire,
94270 Le Kremlin Bicêtre France
e-mail: nelkadhi@club-internet.fr
home page: http://www.epita.fr/~el-kad_n

Abstract. Secure Java applications such as JavaCard applets often rely on cryptography for implementing security functions such as authentication, or the creation of confidential channels. The question addressed by this paper is to provide automated support for the verification of such applications. We describe here our experience in the design and implementation of *StuPa*, a prototype of a static analyzer performing verification of Java cryptographic applets in order to prevent the disclosure of specified sensitive data. We give an overview of the formal models it is based on, particularly those used for modeling cryptographic knowledge. Finally, we review the scope and limitations of the current prototype, and its interest for practical applications. [1]

1 Introduction

JavaCard ([10]) is a simplified subset of Java, which has been lightened and secured to run on smartcards. JavaCard applets are typically used for authentication and electronic commerce applications and have high security requirements. Our objective in this paper is to statically detect potential flaws in an untrusted applet. We are mainly interested in confidentiality properties: while a trusted program is interacting with an untrusted environment, we want to ensure that secret data cannot be disclosed, inadvertently or mischievously. Several formal models for the analysis of cryptographic protocols have been proposed, notably [3] and [13]. These methods allow for the verification of various cryptographic properties such as freshness, authentication and confidentiality. Moreover, some other recent approaches such as [1], [9] and [7], are fully automated.

The point addressed by this paper is the adaptation of these methods to the verification of confidentiality properties of JavaCard applets. Such a task raises several problems. First, these methods generally require a complete specification

[1] This work was partially supported by the TASSC ITEA project

of protocols, including a description of the role of each participant, whereas in the applications we consider, we only possess a Java program. Second, in contrast with these approaches which only apply to dedicated specification formalisms, we need, for practical reasons, to be able to deal with real Java programs in a highly automated way.

The focus of the following work is to derive suitable formal models that comply with these requirements. This task has been done following the methodology of abstract interpretation, as described by [6].

The points addressed in the following are twofold. First, we review in section 2 the formal approach we have designed for the static analysis of confidentiality. This approach has been derived from [3] and is applicable to general cryptographic algorithms. Second, in section 3 we explain how we have adapted this approach to the static analysis of Java. The necessity of dealing with real Java program instead of algorithms specified into a dedicated language raises several concerns we consider below. We explain in sub-section 3.1 how we have dealt with static analysis issues proper to Java such as managing statically unknown values and references. In section 3.2 and 3.3, we consider the problem of identifying and modeling cryptographic actions of Java applets. Finally, our design choices and solutions have been tested and experimented in *StuPa*, a prototype static analyzer. We will briefly summarize the first results of these investigations in section 4.

2 Automated verification of confidentiality properties

To begin with, we designed a convenient formal framework for the static analysis of cryptographic programs. Our approach is an adaptation of Bolignano's method [3], which is oriented toward automated verification of confidentiality. We have modified this method to make it automated and more easily applicable to real programs. The techniques we use to achieve these goals are derived from proof-theoretical results about the evolution cryptographic knowledge. We follow the abstract interpretation methodology in order to translate these results into an automated verification method. The idea of using abstract interpretation in the area of cryptographic protocol verification to get automated procedures has been applied independently in [4] and [9].

Instead of considering Java byte-code at first, we use a simple language of cryptographic actions in order to set up the formal foundations of static cryptographic verification. This language is limited to what is essential and sufficient to describe cryptographic programs: its has simple testing and branching instructions in order to describe program control, and its elementary instructions include cryptographic actions such as equality tests, cryptographic message construction and decomposition, nonce and key generation, and message reception and emission (see table 1 below). Known or public messages are those that could be constructed by an intruder in an untrusted environment using only initially known data and data having leaked from user program. As more and more data get disclosed along program execution, the set of public messages is

$a \in A ::= \epsilon$		*null action*
	$x = y$	*equality test*
|	$\neg\ x = y$	*non-equality test*
|	$x := y$	*assignment*
|	$x := op(x_1, x_2, \ldots, x_n)$	*elementary operation*
|	$y := [x_1; x_2; \ldots, x_n]$	*tuple construction*
|	$detuple(x, [x_1; x_2; \ldots, x_n])$	*tuple read*
|	$x := ?y$	*channel read*
|	$x\ !\ y$	*channel write*
|	$x := encrypt_y^\alpha z$	*encryption*
|	$x := decrypt_y^\alpha z$	*decryption*
|	$decryptfail_y^\alpha z$	*decryption failure*
|	$x := fresh$	*nonce generation*
|	$x, y := keys^\alpha$	*keys generation*

Table 1. *Cryptographic actions*

growing. This set is somehow an approximation of the *cryptographic knowledge* of a hostile intruder. Within this model, we have defined a deduction relation describing when a message is constructible from a set of known messages. What is needed for program verification is an automated procedure to statically compute the evolution of cryptographic knowledge during program execution. We have investigated the formal properties of the deduction relation and derived sound verification procedures from it. A complete description of this approach can be found in [8]. We will only mention that, in this approach, the concrete meaning of cryptographic actions is defined as terms of transformations acting on cryptographic environement made of an environment carrying information about messages content, and of a set containing all disclosed messages describing the state of knowledge of an untrusted intruder. Over that concrete model is defined a static abstract model in order to statically compute the evolution of cryptographic knowledge. Variables are associated to message components, and constraints on these variables express assertions about the structure of messages, and about the set of public data. Sets of constraints approximate cryptographic knowledge and are interpreted as sets of admissible cryptographic environments. Cryptographic actions are interpreted as particular transformers of constraints sets.

We can see how this approach works by considering the following piece of cryptographic protocol (defined in table 2): a message x is read, decrypted into a message z using a secret key k_1, then encrypted into a message w using a secret key k_2, and finally sent. In that example, after stage 1, the constraint $\neg\text{known}(x; k_2)$ means that key k_2 is confidential assuming message x is public, and before stage 3, the constraint $w \approx \{z\}_{k_2}^a$ means that message x is equal to message z encrypted using key u, which is the inverse key of k_1. Table 2 below

action	cryptographic constraints
0. (initial assumptions)	$\kappa_0 = \{is\_simple(k_1),\ \neg known(;k_1),\ is\_simple(k_2),\ \neg known(;k_2)\ \}$
1. $x := ?ic$	$\kappa_1 = \kappa_0 \cup \{is\_simple(k_1),\ \neg known(x;k_1),\ is\_simple(k_2),\ \neg known(x;\ k_2)\ \}$
2. $z := decrypt^a_{k_1} x$	$\kappa_2 = \kappa_1 \cup \{is\_simple(u),\ Inv^a(u,k_1), x \approx \{z\}^a_u\ \}$
3. $w := encrypt^a_{k_2} z$	$\kappa_3 = \kappa_2 \cup \{w \approx \{z\}^a_{k_2}\ \}$
4. $w\,!oc$	$\kappa_4 = \{is\_simple(k_1),\ \neg known(x;k_1),\ is\_simple(k_2),\ \neg known(x;k_2),\ is\_simple(u),\ Inv^a(u,k_1),\ x \approx \{z\}^a_u,\ w \approx \{z\}^a_{k_2}\ \}$

Table 2. *A part of a cryptographic protocol*

shows the evolution of cryptographic constraints resulting from the execution of the protocol.

More generally, this approach has been implemented and tested on programs implementing parts of well-known cryptographic protocols, and it gave sensible results. We use this formal approach in section 3.3 below in order to model cryptographic information in Java programs.

3 Formal models for Java programs

The formal approach of section 2 is ignoring Java-specific problems such as the matching of Java instructions to cryptographic actions, which we consider in sub-sections 3.2 and 3.3 below. In order to set up convenient formal models for Java, we have adapted its former approach to the particular structures of Java Virtual Machine runtime data and values.

On one hand, we have to deal with general issues of Java static analysis such as how to handle unknown values. On the other hand, we have to integrate our general model of cryptographic knowledge in a seamless way into our Java-oriented framework of static analysis. Formal models encoding Java Virtual Machine configurations have to be augmented with cryptographic information, and cryptographic actions have to be identified inside Java programs.

In the following, we explain how we deal with these issues in *StuPa*. As an illustration, we will use a Java implementation of the piece of protocol presented in table 2 above. *StuPa* takes as input a set of Java class-files, and additionnal cryptographic information, and returns as output static information about the cryptographic knowledge at different control points of the program. Table 3 below shows the source code and the byte-code of the *main* method of our example. As additionnal information, we have set the assumptions that the static fields `k1` and `k2` hold confidential keys at their initialization. We also have declared the cryptographic methods `put`, `get`, `encrypt` and `decrypt` so that they are submitted to a particular processing during static analysis. We give some details about this point below.

3.1 Static approximations

The Java Virtual Machine, or JVM, is described in [12], and more formally in [2]. As usual in static analysis, we have to approximate the JVM values which are unknown at compile-time. We make the following choices: unknown JVM numerical values (*int, long, double, float*) are approximated by their type, and unknown references are approximated by their set of possible sites of creation, or *locations*, which are the program control points where they may have been created. The main consequences of these modeling decisions are that actual classes of instances can be resolved accurately, but different instances created at a same control point are modeled as a single abstract instance (corresponding instances fields must be merged).

We also have to make decisions about the modeling of control information. For the sake of simplicity, methods calls are in-lined, which is sufficient in the

```
public static void main() {
  int x_size = 5;    // read incomming message
        // 0 iconst_5
        // 1 istore_0
  int[] x = Env.get(x_size);
        // 2 iload_0
        // 3 invokestatic #8 ⟨Method int get(int)[]⟩
        // 6 astore_1

  int[] z = decrypt(x, k1);    // decipher incomming message
        // 7 aload_1
        // 8 getstatic #9 ⟨Field int k1⟩
        // 11 invokestatic #5 ⟨Method int decrypt(int[], int)[]⟩
        // 14 astore_2

  int[] w = encrypt(z, k2);    // compose outgoing message
        // 15 aload_2
        // 16 getstatic #10 ⟨Field int k2⟩
        // 19 invokestatic #6 ⟨Method int encrypt(int[], int)[]⟩
        // 22 astore_3

  Env.put(w);    // send outgoing message
        // 23 aload_3
        // 24 invokestatic #11 ⟨Method void put(int[])⟩
  return;
        // 27 return
}
```

Table 3. *main method of the Java applet*

absence of recursion, as it is the case in most JavaCard applets. Otherwise, method calls require further standard treatment we don't wish to describe here.

As one can expect, this way we can faithfully model statically-defined instructions: static method call and static fields access and local access on the JVM stack are interpreted without loss of information. Computations on static data are also translated accurately. On the contrary, the abstract interpretation of dynamic instructions induces non-determinism, which arises from branches and method invocation.

3.2 Models of data leakage

In order to be able to state and check confidentiality properties, we must be able to distinguish, inside the program to be analyzed, which part is trusted and allowed to hold secret data, and which part belongs to the untrusted external environment, and toward which no secret information should leak. Note that this requirement differs from our previous cryptographic framework [8], in which such definitions are defined as primitive notions.

Our approach to that point consists in defining a frontier between the user program and its untrusted environment, following an approach inspired from [11]. This is done by declaring a set of trusted classes and by observing the JVM stack. Potential leaks of information are detected as follows. Numerical values are typed and partionned into three sets: surely public messages, possibly confidential messages, and values without cryptographic content (which are not messages). A leak is detected when a trusted method writes a confidential data on a field belonging to an untrusted class, or returns a confidential value to a calling method of an untrusted class, and when a method of an untrusted class gets confidential values as parameters, or reads a confidential data from some field.

For instance, in the example of table 3 above, the definition class of methods `encrypt` and `decrypt` is trusted, whereas the definition class of methods `put` and `get` is not. When an *int* array is passed as argument to the untrusted `put` method, the values contained in the array are checked, and a leak of data is detected for the values which are typed as confidential (values without cryptographic content would produce a type error). This means that this method call will be interpreted as a sequence of *channel write* action on these values, as we will see in sub-section 3.3 below. Similar checks are applied to field access intructions.

In so doing, we define a single confidentiality domain, the limits of which are set at the level of classes (by the distinction made between trusted and non-trusted classes). This approach has appeared to be convenient in practice. One should also note that in order to perform the static analysis of a program, we not only have to supply a set of class-files (to be loaded and launched), but also additional information that specify trusted classes and initially secret data (designated as particular confidential static fields).

This model of data leakage has some similarities to the approach of [11] which considers the dual problem of access to sensitive references. This latter

method consists in using a dedicated type system which assigns a particular typing to references in order to detect access to sensitive references. This approach is applied to a ML-like language with references and functional closures. The type soundness properties of this language are formally investigated in [11] and they allow to use type inference algorithms in order to make the detection. Our approach to data leakage detection is similar in that it also assigns particular typing to sensitive values. But it differs from it in that it uses data flow analysis, rather than type inference, in order to propagate typing information. We have chosen a different approach because, in the case of JavaCard applets, which are somehow more restricted than ML programs, this choice leads to simpler static analysis algorithms.

3.3 Models of cryptographic knowledge evolution

In section 2, we referred to a general model of cryptographic knowledge we have designed for automated verification. We have now to incorporate this formal into a Java-oriented framework of static analysis.

First, we need to define a meaning for cryptographic notions such as messages or cryptographic knowledge within the concrete JVM model of execution. This task raises no particular difficulty. This is done by augmenting the JVM runtime structures with additional cryptographic information. To the JVM global configuration is associated a cryptographic environment in the same way as environments are associated to cryptographic programs in [8]. JVM numerical values are associated with messages showing their cryptographic content.

Second, we have to define what is the cryptographic counterpart of the instructions of this augmented JVM, that is to say how is propagated and modified cryptographic information along execution. This point is less easy, because in the JVM model of execution, there is no simple counterpart to the elementary cryptographic actions defined in our cryptographic model. We proceed as follows: message emission is modeled following the method described in sub-section 3.2 for detecting data leakage; other cryptographic actions such as encryption, decryption, nonce or key creation, and message construction or decomposition are associated to calls to particular library methods. The cryptographic meaning of these methods is hard-coded inside our analyzer as a set of specifications that must be supplied once and for all with each cryptographic API.

Finally, we have to abstract this augmented JVM model of execution into a static abstract model, well-adapted to automated analysis. Following the approach initiated in [8] and presented in section 2, this stage of abstraction poses not particular problem.

We achieve this in our abstract model by associating variables to numerical values having a crytographic content, and by augmenting the JVM configuration state with a set of cryptographic constraints. Admissible cryptographic environments are statically modeled as a set of cryptographic contraints as in the example of table 2 above. During the static analysis of execution, a fresh variable is associated to each new value. Cryptographic constraints on that variable

are generated and transformed along the execution of each instruction which happens to be interpreted as a cryptographic action.

We can get some insights into this approach by looking at the example of table 3.

initial assumptions
$\kappa_0 = \{is\_simple(k_1),\ \neg known(;k_1),\ is\_simple(k_2),\ \neg known(;k_2)\}$
read incomming message 0 iconst_5 1 istore_0 2 iload_0 3 invokestatic #8 ⟨Method int get(int)[]⟩ 6 astore_1
$\kappa_1 = \kappa_0 \cup \{is\_simple(k_1),\ \neg known(x;k_1),\ is\_simple(k2),$ $\neg known(x;k_2),\ \neg known(x;k_2),\ x \approx [x_1;x_2;x_3]\}$
decipher incomming message 7 aload_1 8 getstatic #9 ⟨Field int k1⟩ 11 invokestatic #5 ⟨Method int decrypt(int[], int)[]⟩ 14 astore_2
$\kappa_2 = \kappa_1 \cup \{is\_simple(u),\ Inv^a(u,k_1),\ x' \approx [x_1;x_2;x_3],$ $x' \approx \{z\}^a_u,\ z \approx [z_1;z_2;z_3]\}$
compose outgoing message 15 aload_2 16 getstatic #10 ⟨Field int k2⟩ 19 invokestatic #6 ⟨Method int encrypt(int[], int)[]⟩ 22 astore_3
$\kappa_3 = \kappa_2 \cup \{\ z' \approx [z_1;z_2;z_3],\ w \approx \{z'\}^a_{k_2},\ w' \approx [w_1;w_2;w_3]\ \}$
send outgoing message 23 aload_3 24 invokestatic #11 ⟨Method void put(int[])⟩ 27 return
$\kappa_4 = \{is\_simple(k_1),\ \neg known(x;k_1),\ is\_simple(k_2),\ \neg known(x;k_2),$ $is\_simple(u),\ Inv^a(u,k_1),\ x \approx [x_1;x_2;x_3],\ x' \approx [x_1;x_2;x_3],$ $x' \approx \{z\}^a_u,\ z \approx [z_1;z_2;z_3],\ z' \approx [z_1;z_2;z_3],\ w \approx \{z'\}^a_{k_2},$ $w \approx [w_1;w_2;w_3]\ \}$

Table 4. *knowledge evolution*

For instance, in table 3, at method call `Env.put(w)` near byte-code 23, the int array argument `w` is scanned for leakage detection, and for each leaking value (here, for each value), the associated variables are extracted and cryptographic *write channel* actions are generated. Another illustration is given by method call `encrypt(z, k2)` near bytecode 19: in that case, the values contained in the array argument `z` are fetched, the sets of their associated variables z_1, z_2 and z_3 are read, and the following cryptographic actions are generated: first, a tuple construction of the message to

be encrypted: $z' := [z_1; z_2; z_3]$; second, an encryption of that message (using a fresh variable w as result): $w{:=}\ encrypt^a_{k_2}\, z'$; and third, a tuple reading of the resulting message (using fresh variables w_1, w_2 and w_3 for the contained values): $w := [w_1; w_2; w_3]$. (The tupling and detupling actions are necessary in order to set the lengths of the arrays taken as argument and returned as result.) Table 4 shows the kind of cryptographic information that *StuPa* can deduce from the static analysis of the applet. In particular, we can check data confidentiality by tracking constraints $\neg known(...; ...)$ and $\neg old(...; ...)$. The example above shows that the keys k_1 and k_2 remain confidential up to the end of the execution of the protocol.

4 First implementation results

We have reviewed above the underlying formal models used as basis for the design of *StuPa*, an automated tool for the static verification of confidentiality of Java cryptographic applets.

At the current stage of development, a prototype is working, and is under experimentation. Up to now, it has been tested on small cryptographic applets such as one implementing the user's role in the Yalahom protocol described in [5]. On these examples, our tool yields good results in terms of leak detection and verification of confidentiality. Experiments on larger applets are on the way. The main task required for carrying them is to incorporate cryptographic specifications of the JavaCard API into the analyzer.

After some usage, the main limitations that appear are related to loss of information in the process of static analysis. Some limitations result from design decisions. As it is oriented toward the verification of JavaCard applets, our formal model doesn't support features of the Java language which are un-used in this Java subset, such as threads and reflection. Other sources of information loss are related to the existing trade-off between accuracy of analysis and its cost. For instance, we could get more information by unfolding iterations more. For those cases, it is possible to fine-tune the analyzer.

5 Conclusions

An innovative aspect of *StuPa* is undoubtedly the way it handles cryptographic knowledge: this tool rests on a formal framework that provides both an accurate description of cryptography, and a well-fitted model for automated analysis. It enables us to formally relate Java implementations of cryptographic mechanisms to their design requirements, and to verify their conformance in an automated way.

Much care has been taken in achieving a sound design, in accordance with the methodological principles of the abstract interpretation framework. In particular, formal foundations of the model of cryptographic knowledge have been extensively investigated ([8]).

StuPa sets the focus of static analysis on JavaCard-compliant applets, for which first experiments gave conclusive results. The small size of these applets

and the restricted subset Java they are implemented on mostly account for this success. For more general Java programs, more look is needed to deal with recursion (which poses no theoretical difficulties), thread and reflection (which are harder to deal with).

Nevertheless, even as limited to JavaCard, this approach seems promising, because there are today are high needs of formal verification on JavaCard applets, which have high security requirements.

References

1. Monniaux, D. Abstracting cryptographic protocols with tree automata. In *Static Analysis* (1999), A. Cortesi and G. Filé, Eds., vol. 1694 of *Lecture Notes in Computer Science*, Springer, pp. 149–163.
2. Bertelsen, P. Dynamic semantics of Java bytecode. In *Workshop on Principles of Abstract Machines* (Pisa, Italy, Sept. 1998).
3. Bolignano, D. An approach to the formal verification of cryptographic protocols. In *3rd ACM Conference on Computer and Communications Security* (New Delhi, India, Mar. 1996), C. Neuman, Ed., ACM Press, pp. 106–118.
4. Bolignano, D. Using abstract interpretation for the safe verification of security protocols. In *Electronic Notes in Theoretical Computer Science* (2000), M. M. Stephen Brookes, Achim Jung and A. Scedrov, Eds., vol. 20, Elsevier Science Publishers.), pp. 77–87.
5. Burrows, M., Abadi, M., and Needham, R. A logic of authentication. *ACM Transactions on Computer Systems 8*, 1 (Feb. 1990), 18–36.
6. Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages, Los Angeles* (New York, NY, 1977), ACM, pp. 238–252.
7. Lowe, G. Casper: A compiler for the analysis of security protocols. In *10th IEEE Computer Security Foundations Workshop (CSFW '97)* (Washington - Brussels - Tokyo, June 1997), IEEE, pp. 18–30.
8. Elkadhi, N. Automatic verification of confidentiality properties of cryptographic programs. *Networking and Information Systems* (2001), pp. 4–15, Available at url: `http://www.epita.fr:8000/~el-kad_n/Hermes.ps`
9. Goubault-Larrecq. A method for automatic cryptographic protocol verification. In *SPDP: IEEE Symposium on Parallel and Distributed Processing* (2000), ACM Special Interest Group on Computer Architecture (SIGARCH), and IEEE Computer Society.
10. Sun Microsystems, Inc. *The JavaCard 2.2.1 Platform Specification.* Palo Alto/CA, USA, May 2000.
11. Leroy, X., and Rouaix, F. Security properties of typed applets. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California* (New York, NY, Jan. 1998), ACM, pp. 391–403.
12. Lindholm, T., and Yellin, F. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, Reading, MA, USA, Jan. 1997.
13. Meadows, C. Applying formal methods to the analysis of a key management protocol. *Journal of Computer Security 1*, 1 (1992), 5–36.

6 Authors background and expectations

Nabil EL KADHI is professor of computer science at Paris EPITA/EPITECH. Among his research interests are the security of cryptographic protocol and the static analysis of program confidentiality.

He is the author of a Phd thesis on the "automatic verification of confidentiality properties of cryptographic programs".

Pierre BOURY is research engineer at GIE Dyade, a common BULL-INRIA subsidiary. His main research interests are formal methods applied to software engineering. He has recently been involved in a project of cryptographic protocols verification using the Coq proof assistant, and is today developing static analysis tools for the automated verification of Java cryptographic applets.

Our expectations toward the Workshop on Formal Techniques for Java Programs are to share ideas about formal verification of security properties in the domain of Java, and particularly to bring our contribution to the field of automated verification of cryptographic security.