An Information Flow Verifier for Small Embedded Systems*

Dorina Ghindici, Gilles Grimaud, and Isabelle Simplot-Ryl

L.I.F.L. CNRS UMR 8022
Université de Lille I, Cité Scientifique
F-59655 Villeneuve d'Ascq Cedex, France
{ghindici,grimaud,ryl}@lifl.fr

Abstract. Insecurity arising from illegal information flow represents a real threat in small computing environments allowing code sharing, dynamic class loading and overloading. We introduce a verifier able to certify at loading time Java applications already typed with signatures describing possible information flows. The verifier is implemented as a class loader and can be used on any Java Virtual Machine. The experimental results provided here support our approach and show that the verifier can be successfully embedded. As far as we know, this is the first information flow analysis adapted to open embedded systems.

Keywords: information flow, type checking, confidentiality, class loading.

1 Introduction

As the use of Java-enabled embedded systems such as smart cards, mobile phones and PDAs is growing, they are being associated with security since they provide a partial solution to the need for personal identification and non-repudiation. These devices evolve towards an open, multi-applicative environment supporting dynamic class loading and unloading.

Confidential data manipulated by such systems must be protected and accessible only by authorized users or programs. In a small open embedded system, where application may share code (e.g Api) or collaborate to offer better services, insecurity may stem from the code itself or from the code shared with some malicious untrusted software.

In order to enforce security, the Java Virtual Machine (Jvm) [17] and the Java Runtime Environment provide different mechanisms. For example, the bytecode verifier [16] uses static analysis to ensure that applications comply with the Java type system rules even for small systems [8,19], while the sandbox model is a dynamic mechanism, which enforces security by isolating applications. Java access modifiers (e.g. *public*, *private*, *protected*) express data accessibility for the Java language. Existing Java mechanisms control information access, and so

* Funded by the Mosaïques project, CPER TAC 2005-2008 & FEDER.

D. Sauveron et al. (Eds.): WISTP 2007, LNCS 4462, pp. 189–201, 2007.

they are not adequate in addressing data propagation. Ad-hoc mechanisms, like information flow verification, must be added to guarantee safe data propagation.

Despite the considerable amount of work on information flow achieved in the past decades, the information flow based enforcement mechanisms have not been widely used and applied in practice [26]. A survey on language-based security and information flow is presented in [20]. Most of the contributions in this area are based on static analysis [2,7,10,18,25] and on the type-based approach [3,13,22], where a type system is used to check secure information flow.

Low-level languages are taken into consideration only in few papers [3,15], while the Java language is rarely specified. In [4], only a small subset of the Java language is taken into consideration, while in [9], a compositional information flow analysis was implemented for mono-threaded Java bytecode. In [2], authors propose a static analysis similar to standard type verification used for Java bytecode. JavaCard features are considered in [5] and [11]. JFlow [18] is a powerfull tool, implemented as an extension of the Java language and structured as a source-to-source translator. It adds reliability to software implementation, but not to deployment and linking on a platform. The Pacap framework [5] involves a technique to verify interactions for Java enabled smart-cards, but the verification relies on the call graph, so it cannot be trusted in a Java/javacard open environment.

Unfortunately, the previous models focus on correctly checking information flow statically and do not address the challenge raised by an open computing environment.

In this paper, we propose an efficient model for detecting illegal information flows. Our model was successfully applied on small, open, Java-enabled systems. Our goal is to enforce data confidentiality for standard Java mobile code. In order to address the challenge raised by an open system, we enforce security properties at load time by performing a static analysis. Our implementation [23] works directly on Java bytecode and includes support for dynamic class loading and overloading.

The rest of the paper is structured as follows: Section 2 introduces some aspects of information flow and our approach. In Section 3 we present how we enforce confidentiality at load time on an embedded system. Section 4 describes how the information flow verifier can be integrated within a dedicated class loader in the KVM, but also how it can be used on any Jvm (e.g. JavaCard 3.0) as a user-defined class loader. Section 5 presents experimental results, while Section 6 summarizes our contributions.

2 Information Flow Analysis

2.1 General Aspects

Information flow stands for data propagation in a program. There are information flows arising from assignments (direct flows), from the control structure of a program (implicit flow), etc. For example, the code `p=s` generates a direct flow from `s` to `p`, while `if(s) then p=1 else p=0` generates an implicit flow. If `s` contains a secret, confidential value, and `p` a public, observable value, then the two examples are insecure and generate an illicit information flow, as confidential data can be induced by the reader of `p`. In literature, confidentiality is often seen as a non-interference [7,25] problem, as *public* outputs cannot depend on *secret* inputs. More exactly, for any initial value of an input *secret* variable, the values of *public* outputs do not change.

We target open (embedded) systems, allowing dynamic class loading, overloading and all the Java features, and supporting multi-applications sharing code. In this context, the insecurity for a class `A` may arise from the fact that `A` invokes an untrusted method `B.m` and it passes as argument a secret value. The system cannot guarantee that `B` does not make available the secret data. As our system must fit the Java dynamic class loading paradigm, we cannot use traditional approaches verifying the non-interference on the call-graph of a single application. In order to deal with openness, we perform a compositional analysis, computing for each method a stand-alone signature. The signature of a method is independent of the context under which the method is called. It contains the flows, potentially generated by the execution of the method, between elements that survive method execution: the method parameters, the method return value, an abstract value for the static world, one for exceptions, one for input/output channels. One "type" is associated with every element reflecting the flows generated by the method between this element and the others. Based on the knowledge of the flows, an application can check its own security policy. Thus, direct flows inside methods will be detected by traditional analysis while flows generated by interactions between methods will be detected by composition of the methods signatures. (Implicit flow inside methods is a natural extension of our approach, as the low complexity of the existing algorithms is promising for an embedded verification.)

2.2 Algorithm

We propose a model in which, as in classical information flow, each field is annotated by a security level, *secret* or *public*: a *secret* field should not be made accessible through information flow to unauthorized parties. Tracing all the fields of an object is expensive in time and memory and is not always possible when the calling context is not available. Moreover, imposing some kind of "subtyping" of signatures constrains the use of overloading. So, we split each object in only two parts, a secret part and a public part. The secret part of an object o, denoted by o^s, stands for all access paths starting from o which contain at least one field having the security level *secret*, while the public part, denoted by o^p is the complementary. Experimental results showed that our simplifying assumptions are reasonable in practice.

Considering our split of objects and the dichotomy of Java types (elementary types and object types), the links between two elements a and b have the form $a^{\wp(p,s)} \xrightarrow{v/r} b^{\wp(p,s)}$, where v denotes a value link, r a reference/alias link, s and p the secret or public part, while $\wp(p,s)$ denotes subsets of $\{p,s\}$. As a reference link includes the value link between the same elements, and as the *public* or *secret* part are included in the entire element, we can define an order relation between flows. Using this partial order relation, we obtain a lattice of links that contains 80 possible flows between two elements. The bottom of the lattice is represented by an empty set, meaning that there is no flow of information between a and b, while the top of the lattice is represented by $\{a^{p,s} \xrightarrow{r} b^{p,s}\}$, meaning that there is a reference link (alias) between the secret and public part of a and the secret and public part of b. More details on the type system and the lattice of links can be found in Appendix A.

Let's consider a class C having a *secret* field `s`, a *public* field `p` and a method `m`. Fig. 1 presents the signature of `m` at each point of the program, considering that the external method `foo` contains a value link from the return of the method, denoted by R, to the first parameter of the method ($R \xrightarrow{v} p_1$).

	`void m(int x, A a) {`		
1		`iload_1`	
2	`if(x>0)`	`ifle 6`	
3		`aload_0`	
4		`iload_1`	
5	`this.s = x;`	`putfield C.s`	$S_m = \{this^s \xrightarrow{v} x\}$
6		`aload_0`	
7		`aload_2`	
8		`iload_1`	
9		`invokevirtual A.foo`	
10	`this.p = a.foo(x);`	`putfield C.p`	$S_m = \{this^s \xrightarrow{v} x, this^p \xrightarrow{v} x\}$
11	`}`	`return`	with $S^e_{foo} = \{R \xrightarrow{v} p_1\}$

Fig. 1. Example

In order to ensures threads-safety, the abstraction for static elements has the default security level *public*, as all the *secret* data linked to static attributes and susceptible to be accessed through different threads are considered leaky

To compute methods signatures, we perform for each method an intra-method static abstract interpretation relying on a classical operational semantics composed of a set of transformation rules. The abstract values are represented by elements composing the signature of the method, and some other internal values needed to correctly analyze the method. The analysis does not rely on the call graph: the interpretation of an *invoke* bytecode consists of applying the signature of the called method to the signature of the calling method.

Each instruction has associated an abstract state representing the state before executing the instruction. The state contains the local operand stack, the local variables, and the current signature of the method at an execution point. This state contains the union of all possible states under which the associated bytecode can be executed. The control flow structure of the Java bytecode dictates an iteration on the set of instructions for each method. At each iteration, the current bytecode is abstractly interpreted and the resulting state is merged with the state already associated with its successors. For *invokevirtual* bytecodes, when the exact type of the called object cannot be statically determined, we take into consideration a global signature which is the union of all possible signatures for the desired method implemented in the class hierarchy derived from the static type of the object. Since the number of abstract values is finite and we perform merge operations, a fixed point is reached.

The existence of recursive and inter-depended methods dictates an incremental inter-method analysis, starting with the set of empty signatures and iterating on a set of classes until a fixed-point has been reached. This allows us to obtain more precise results.

Noninterference is too restrictive for common applications such as cryptographic functions, where outputs often depend on secret outputs. However, one should not be able to derive the secret from the output. To handle this and other intentional release of secret data, the proposed systems allows to manually annotate trusted methods with the desired signature. A more precise approach would be to include a mechanism for declassification [21,12] in order to specify what information can be released and where.

3 Information Flow Verifier

In the previous section, we presented a compositional information flow analysis complying with the Java dynamic class loading paradigm. But experimental results, as depicted in Fig. 4, show that at least 3 iterations must be performed on the set classes and an average of 1.5 iterations on the set of bytecodes for each method. Therefore, the analysis is already expensive for a normal system and impracticable for a device having limited resources.

In the context of small objects, a technique known as "Lightweight bytecode verification" (LBV) [19] has been developed for Java bytecode type verification. This technique, closely related to proof-carrying code [6], lies on the simple idea that it is easier to verify a result already computed. A small device can verify code received from an untrusted source without relying on a third party even if it has not enough power to compute the proof itself. It is based on two phases: an external phase which computes the type correctness and annotates the bytecode with some proof elements, and an embedded phase, which verifies, at loading time, the annotations obtained during the external phase. The verification operation is linear in code size and uses constant memory. The off-board analysis and the proof can be computed by any device or tool, as the small device can verify the code it receives without relying on the external device. LBV relies on the lattice structure of types and on unification operations on this lattice. The lattice of links allows us to use this technique in our context. While LBV checks explicit Java types, our algorithm has to infer information flow links. We have to deal with type inference and with signature management.

3.1 Signature Computation

The verification process is performed while loading a Java class. In order to ease verification, we ship with each class C some proof: the state of the JVM for each target instruction in each method, the signatures of methods invoked in C, the security levels of fields used in C. The proof elements are defined as new attributes of the `class` file structure, so the annotated classes can be loaded by any JVM, even by those not enforcing information flow security properties.

Due to limited resources of embedded systems, the size of proof elements must be as small as possible. As the lattice of links contains 80 possible flows, we chose a binary and compact solution on 1 byte to encode the links. This solution allows simple manipulation operations. For example, adding a new link to a signature requires only a binary logical operation. Moreover, the signatures within the states of the JVM for each target instruction are encoded incrementally: the first signature is encoded, while the subsequent signature is defined by the the flows added or deleted in the previous signature. Experimental results showed that signatures have few changes from one label to another.

The verification consists in a sequential interpretation of bytecodes of each method of the class. When a target bytecode is found, the current state of the JVM must be compatible with the proof corresponding to the target bytecode: if the compatibility is not tested, the class is rejected; otherwise the verification is carried on using the proof as the current state of the JVM. Given two states A and B of the verification process, A is compatible with B if the stack and local variables are compatible (state B contains at least all the elements in A) and the two signatures, S_A and S_B, are compatible. The stack comparison is possible, as we assume that the bytecode was already checked by the JVM verifier and thus it is well typed. A signature S_A is compatible with S_B if S_A contains at least all the links present in S_B, according to the lattice of method signatures, which is a natural extension of the lattice of links defined previously.

Dead code is ignored by the external analysis and thus not annotated. In order to deal with this situation, when a label bytecode without any proof annotation is found, we can assume it is the beginning of a block which is never executed. In this case, all the bytecodes following the label are ignored, until we meet a label with a proof. If the label without a proof is not the start of a dead block, then the class is rejected when the compatibility of predecessors instructions with the proof of the label is tested.

The embedded verification has the advantage that each instruction is interpreted only once and so it is linear in time with the code size. Moreover, the proof is used only during the verification and not stored in the system. Only the final signature of each method is kept on board. Another advantage is that each class is verified only once, even the code shared by many applications, as the signatures are kept on board in a dictionary. If the type inference of method signature fails, the class is rejected. If the type inference succeeds, we must ensure that the signatures used during validation fit within the system.

The analysis guarantees noninterference for loaded classes. All the possible flows from secret to public data are detected and present in the signatures.

But, due to our simplifying assumptions, we might detect false flows. Practical experiments showed that this situation does not occur very often.

3.2 Signature Management

Classes are loaded one by one. Once a class is loaded, the validated signatures are kept in a dictionary. In order to validate a class C at loading time, we load with C the signatures of all methods invoked in C (called "external methods").

When we load the class C from the example in Fig. 1, we will also load the signature S^e_{foo} of the method `foo` in the class A. If the class A has already been loaded, the external signature S^e_{foo} will be ignored and the signature of foo from the dictionary will be used while verifying C. If the class A has not been yet loaded, S^e_{foo} will be used while analysing C. If C is accepted, the signature S^e_{foo} will be kept on board into a temporary dictionary until the class A is loaded.

Let's assume that A is loaded later and the method foo has the signature S_{foo}. A will be accepted only if the signature S_{foo} is compatible with S^e_{foo}. The external signature S^e_{foo} should contain at least all the links from the loaded signature S_{foo}, otherwise the previous verification of C is not correct. If the external signature contains fewer links than the loaded one, it is acceptable, as long as we do not miss any information leakage. If the class A is certified and loaded on the system, S^e_{foo} and all the external signatures for A previously loaded are erased from the temporary dictionary.

There are different possible scenarios. We now consider the following:

load class C
  external method A.foo with $S'_{foo} = \{R \xrightarrow{v} p_1\}$
  store S'_{foo} in the temporary dictionary
load class D
  external method A.foo with $S''_{foo} = \{R \xrightarrow{v} p_1, this^s \xrightarrow{v} p_1\}$
  store infimum(S'_{foo},S''_{foo})=$\{R \xrightarrow{v} p_1\}$ in the temporary dictionary
load class A
  A.foo with signature S_{foo}

We load two classes C and D, and each one claims a different external signature for $A.foo$. As to validate the class C we use S'_{foo}, and to validate D we use S''_{foo}, the real signature S_{foo} should be compatible with S'_{foo} and S''_{foo}. All the flows in S_{foo} should be in S'_{foo} and S''_{foo}, which means that S_{foo} should be compatible with infimum of S'_{foo} and S''_{foo} according to the lattice of method signatures. So when we have different external signatures for the same method, we keep the infimum in the temporary dictionary.

The correctness of a signature depends also on the security levels of used fields. To have access to security levels of external fields of a class, we use a procedure similar to the one used to load the external methods. Two fields are compatible if they have the same security level. We also check the compatibility of loaded signatures with the global signatures belonging to extended classes.

4 The Verifier as a User-Defined Class Loader

The loading process in a JVM is performed by the class loaders. The standard JVM deals with multiple class loaders, hierarchically organized, and supports user-defined class loaders. The KVM virtual machine [24] does not support user-defined class loaders and has a single built-in class loader that cannot be overridden or replaced by the user.

We built a verifier that can be run on any JVM. It can be built in the single class loader of KVM or installed as a user-defined class loader for a standard JVM. The embedded JVM [1,14,24] are evolving towards the standard Java language, and therefore towards a multiple class loader hierarchy. The recently presented JAVACARD 3.0 does the same. We describe now how the verifier can be used as a plug-in within a standard JVM to validate annotated bytecode.

Scl_2 loads class C
  external method Scl_2.A.foo with signature S^2_{foo}
Scl_1 loads class D
  external method Scl_1.A.foo with signature S^1_{foo}
Scl_1 loads class A
  Scl_1.A.foo with signature S_{foo}

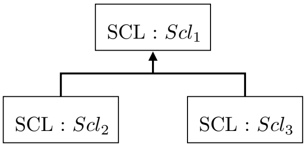

Fig. 2. Loading example in a *SafeClassLoader* hierarchy

Applications implement subclasses of ClassLoader in order to extend the manner in which the JVM dynamically loads classes. Class loaders may typically be used to check security properties. The verifier was implemented as a subclass of the *ClassLoader* class provided by the Java API, named *SafeClassLoader*. Certifying the underlying information flow of an application requires the instantiation of a *SafeClassLoader* with which the application should be loaded.

In the JVM delegation model, class loaders are arranged hierarchically in a tree, with the bootstrap class loader as the root of the tree. Each user-defined class loader has a "parent" class loader. When a load request is made by a user-defined class loader, that class loader usually first delegates the parent class loader, and only attempts to load the class itself if the delegate fails to do so. A loaded class in a JVM is identified by its fully qualified name and its defining class loader. This is sometimes referred to as the runtime identity of the class. Consequently, each class loader in the JVM can be said to define its own name space. In the same manner, each *SafeClassLoader* defines its own dictionary containing the signatures of loaded methods.

Let's consider a hierarchy containing three *SafeClassLoader*s, Scl_2, Scl_3 and their parent Scl_1, and the scenario in Fig. 2. Class loader Scl_2 requests to load class C. At first, it delegates its parent class loader, Scl_1, to load C. If the delegation fails, Scl_2 attempts to load the class by itself. While loading C, Scl_2 tries to find the signature of $A.foo$: it first searches in its dictionary, and if the search fails, it delegates the search to its parent, which repeats the procedure. If the parent also fails to find the signature, external signature S^2_{foo} is used while validating C and stored in the temporary dictionary. Class loader Scl_1 loads a class D also containing an external signature for $A.foo$. The external signature S^1_{foo} is stored in the temporary dictionary and is associated with Scl_1.

Finally, class loader Scl_1 attempts to load class A. Let S_{foo} be the verified signature of foo. Class A will be loaded by Scl_1 if and only if S_{foo} is compatible with the external signatures for foo in the current class loader (Scl_1) and with the external signatures in class loaders that can delegate Scl_1. In our case, S_{foo} must be compatible with S^1_{foo} and S^2_{foo}. Otherwise, class A is rejected.

In order to verify this kind of compatibility, external signatures must be accessible to all class loaders. This is why we implemented an unique temporary dictionary which is used by all class loaders. The example showed how the *SafeClassLoader* extends the delegate model to the look up of a signature of a method. The same search process is extended to the look up of the security level of a field.

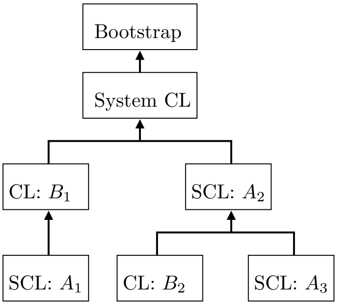

Fig. 3. Class Loader hierarchy example

We presented so far the case where all the class loaders in the hierarchy have the type *SafeClassLoader*. Actually, the hierarchy contains different types of class loaders. As shown in Fig. 3, the bootstrap class loader loads the classes from the JVM, as well as extensions to the JDK. The system class loader loads all the classes provided by the classpath. In the end, we have several additional class loaders, where *SCL* defines a *SafeClassLoader* and *CL* any other type of class loader.

As a consequence, we must take into consideration the validation of classes loaded by any class loader. Let's consider that A_1 loads a class C that invokes a method of another class D already loaded by the parent B_1. As B_1 is not a *SafeClassLoader*, the classes it has loaded have not been validated at loading time. To ensure security for C, *SafeClassLoader* A_1 will try to retrieve, using the *getResourceAsStream* method, the .class files of the classes loaded by its parent and to verify the announced signatures. If the streams cannot be found, or they do not contain information flow attributes, or the signatures are not compatible with the announced ones, A_1 rejects class C. Otherwise, the signatures of classes belonging to a non-*SafeClassLoader* are stored in a special dictionary, named "system dictionary". The look up for a signature in a class loader is performed in its dictionary, if the class loader is a *SafeClassLoader*, and otherwise in the system dictionary.

In order to support any JVM, we do not interfere while the Bootstrap and System class loaders load the JVM and classpath classes, and thus we consider their signatures as part of our trust computing base.

5 Experimental Results

This section describes the results of experiments run on some significant benchmarks such as Dhrystone, a well known benchmark for embedded systems, The Fast Fourier Transform (FFT), a common signal processing application, crypt (a data encryption algorithm) and Pacap [5], an electronic purse case study for information flow checking (for which we detected the same illicit flows as in literature). We ran the experiments using a Java Runtime Environment, Standard Edition (build 1.5.0_09), on a Linux system running on a Intel(R) Pentium(R) M processor 2.13GHz with 1Gb memory.

			Off board analysis					On board verification				
Benchmark	Classes	Methods	Class iterations	Bytecode iterations	Analysis time (s)	Average memory (Kb)	Maximum memory (Kb)	Execution time CL (ms)	Execution time SCL (ms)	Verification time SCL (ms)	Average memory (Kb)	Maximum memory (Kb)
Dhrystone	5	21	3	1.35	3.1	2.66	35.80	111	373	310	0.36	2.05
fft	2	20	3	1.55	3.2	1.50	7.86	63	175	161	0.39	1.80
_201_ compress	12	43	3	1.84	4.4	2.02	20.68	321	522	251	0.37	2.31
_200_check	17	109	4	1.18	8.7	3.87	34.45	129	883	794	0.58	3.51
crypt	2	18	3	1.32	4.1	2.91	20.58	46	268	222	0.56	3.68
lufact	2	20	3	1.75	3.4	3.90	23.22	537	876	303	0.45	1.25
raytracer	12	72	5	1.53	4.6	1.30	25.10	80	440	422	0.39	1.80
Pacap	15	92	4	1.05	4.7	3.00	92.68	30	281	275	0.38	3.19

Fig. 4. Off board analysis and on board verification measurements

First, we ran the external application computing the information flow signatures and annotating the classes (Fig. 4, Off board analysis) in order to find out how the algorithm performs in practice. We measured the number of iterations for the inter-method analysis (iterations on a set of classes), the iterations for the intra-method analysis (iteration on each meathod's instructions set) and the time needed to perform the analysis. The results showed that the computation algorithm is quite expensive in terms of time complexity: in average, we need 3 iterations on the set of classes, 1.5 iterations on the instruction set and 4.5s

Benchmark	Initial class size (Kb)	Annotated class (Kb)	Signatures (%)	Labels proof (%)	External methods (%)	External fields (%)
Dhrystone	8.2	11.9	3.17	23.07	2.42	0.20
fft	6.8	11.3	4.53	48.47	5.17	0
_201_ compress	20.1	28.3	3.79	23.36	4.21	0.33
_200_check	46.3	80.3	4.62	57.25	3.82	0.05
crypt	7.0	12.3	6.04	58.34	4.66	0.19
lufact	9.3	14.3	3.37	43.00	2.06	0.40
raytracer	24.0	33.4	1.41	16.62	5.51	0.63
Pacap	26.8	36.9	5.71	18.37	3.46	0.38

Fig. 5. Size of annotations

for each application. For the JVM *spec* benchmarks, we performed the library analysis before carrying out the experiments.

Secondly, we loaded the annotated applications generated by the off board analysis (Fig. 4, On board analysis). In order to find out how the JVM loading process is hampered by our verification, we measured the execution time in two cases: with (SCL) and without (CL) information flow verification. We observed that the verification implies an average execution time 3 times as large as the standard one. But the information flow verification is performed only once, at loading time, so any subsequent running of the applications is not hindered. Moreover, the average verification time (342.25ms) is more than 10 times smaller than the average analysis time (4.25s). As expected, the verifier performs much faster than the computation algorithm.

Lastly, we measured the size of the proof and the signatures loaded with the code (Fig. 5). The proof, the external methods and external fields represent 39.73% of the total size of initial .class files. This data is used only during the verification process, at loading time, and it is not stored on the device, so its size does not have a significant impact on the embedded system. The signatures, which are stored in the dictionary and kept in the system, make up only 4.01% of the initial .class size, an acceptable overhead.

6 Conclusion

Confidentiality represents a real concern in embedded systems manipulating sensitive data. Practical information flow models are almost non-existent, despite the quality of the underlying theory. As the algorithm for detecting illegal information flows in an application has a high complexity, we propose a lightweight verification for embedded systems. The information flow is certified at loading time, using some proof elements previously computed and shipped with the code. The information flow verification is performed in linear time and uses almost constant memory. Experimental results conducted for the external analysis and for the embedded verification support our approach. The time penalty and the memory consumption introduced by the verifier are acceptable. We think that we can cut by at least 50 percents the size of the elements embedded within the code by modifying the encoding of proof and signatures.

References

1. AONIX INC. Perc products.
2. AVVENUTI, M., BERNARDESCHI, C., AND FRANCESCO, N. D. Java bytecode verification for secure information flow. *SIGPLAN Not. 38*, 12 (2003), 20–27.
3. BARTHE, G., BASU, A., AND REZK, T. Security types preserving compilation: (extended abstract). In *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004* (2004), vol. 2937 of *Lecture Notes in Computer Science*, Springer, pp. 2–15.

4. Barthe, G., D'Argenio, P., and Rezk, T. Secure Information Flow by Self-Composition. In *Computer Security Fundation Workshop (CSFW'17)* (2004), IEEE Press, pp. 100–114.
5. Bieber, P., Cazin, J., Girard, P., Lanet, J.-L., Wiels, V., and Zanon, G. Checking secure interactions of smart card applets: extended version. *J. Comput. Secur. 10*, 4 (2002), 369–398.
6. Colby, C., Lee, P., Necula, G. C., Blau, F., Plesko, M., and Cline, K. A certifying compiler for java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation* (New York, NY, USA, 2000), ACM Press, pp. 95–107.
7. Denning, D. E., and Denning, P. J. Certification of programs for secure information flow. *Commun. ACM 20*, 7 (1977), 504–513.
8. Deville, D., and Grimaud, G. Building an 'impossible" verifier on a Java card. In *Proc. 2nd USENIX Workshop on Industrial Experiences with Systems Software (WIESS'02)* (Boston, USA, 2002).
9. Genaim, S., and Spoto, F. Information Flow Analysis for Java Bytecode. In *Proc. of the Sixth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)* (Paris, France, January 2005), R. Cousot, Ed., vol. 3385 of *LNCS*, Springer-Verlag, pp. 346–362.
10. Ghindici, D., Grimaud, G., and Simplot-Ryl, I. Embedding verifiable information flow analysis. In *Proc. Annual Conference on Privacy, Security and Trust* (Toronto, Canada, 2006), pp. 343–352.
11. Hansen, R. R., and Probst, C. W. Non-interference and erasure policies for java card bytecode. In *6th International Workshop on Issues in the Theory of Security (WITS '06)* (2006).
12. Hicks, B., King, D., and McDaniel, P. Declassification with cryptographic functions in a security-typed language. Tech. Rep. NAS-TR-0004-2005, Network and Security Research Center, Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA, USA, May 2005.
13. Hunt, S., and Sands, D. On flow-sensitive security types. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006* (2006), ACM, pp. 79–90.
14. Java In The Small. http://www.lifl.fr/POPS/JITS/.
15. Kobayashi, N., and Shirane, K. Type-based information flow analysis for low-level languages. *Computer Software 20(2)* (2003), 2–21.
16. Leroy, X. Java bytecode verification: Algorithms and formalizations. *J. Autom. Reason. 30*, 3-4 (2003), 235–269.
17. Lindholm, T., and Yellin, F. *Java Virtual Machine Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
18. Myers, A. C. Jflow: practical mostly-static information flow control. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1999), ACM Press, pp. 228–241.
19. Rose, E., and Rose, K. H. Lightweight bytecode verification. In *Workshop "Formal Underpinnings of the Java Paradigm", OOPSLA'98* (1998).
20. Sabelfeld, A., and Myers, A. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications, 21(1), 2003. 21*, 1 (january 2003).
21. Sabelfeld, A., and Sands, D. Dimensions and principles of declassification. In *CSFW '05: Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 255–269.

22. Smith, G., and Volpano, D. Secure information flow in a multi-threaded imperative language. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1998), pp. 355–364.

23. STAN - STatic Alias aNalyser. http://www.lifl.fr/~ghindici/STAN/.

24. Sun Microsystem. Connected Limited Device Configuration and K Virtual Machine, `http://java.sun.com/products/cldc/`.

25. Volpano, D., Irvine, C., and Smith, G. A sound type system for secure flow analysis. *J. Comput. Secur. 4*, 2-3 (1996), 167–187.

26. Zdancewic, S. Challenges for information-flow security. PLID'04 The First International Workshop on Programming Language Interference and Dependence, August 25 2004, Verona, Italy, August 2004.

A The Type System

For each element a from the set of abstract values we define the flow relation as a tuple composed of four elements: the security level of a (in $\wp(p,s)$), the type of flow (v or r for *value* or *reference*), the element to which a points to and its security level. The type associated with a is the union of all possible flows from a to b. For example, a *value* link from the *public*part of a to the *secret* part of b corresponds to the type (p, v, b, s). For convenience, we will denote this flow by $a^p \xrightarrow{v} b^s$.

If both public and secret parts of an element can be accessed, the whole element can be accessed. Thus, a link to the entire element ($b^{p,s}$) is greater than the same link to only one part of the element(b^s, b^p). Moreover, having access to a reference means having access to all its values. Thus, the *value* link is included in the *reference* link. Using this partial order relations, we obtain a lattice of links (Figure 6) having union as upper bound computation and inclusion as order relation.

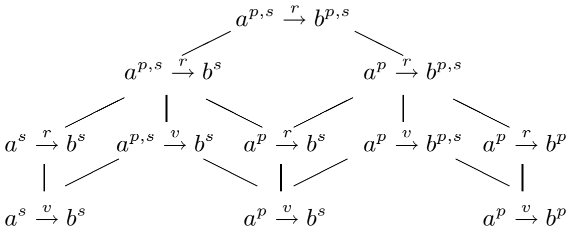
$a^{p,s} \xrightarrow{r} b^{p,s}$
$a^{p,s} \xrightarrow{r} b^s$
$a^p \xrightarrow{r} b^{p,s}$
$a^s \xrightarrow{r} b^s$
$a^{p,s} \xrightarrow{v} b^s$
$a^p \xrightarrow{r} b^s$
$a^p \xrightarrow{v} b^{p,s}$
$a^p \xrightarrow{r} b^p$
$a^s \xrightarrow{v} b^s$
$a^p \xrightarrow{v} b^s$
$a^p \xrightarrow{v} b^p$

Fig. 6. Extract of the lattice of links