Secure Self-Certified Code for Java

M. Debbabi[†‡], J. Desharnais[†], M. Fourati[†], E. Menif[†],
F. Painchaud[†], and N. Tawbi[†]

[†]LSFM Research Group, Computer Science and Software Engineering Department,
Laval University,
Quebec, Canada
[‡]Panasonic Information and Networking Technologies, Princeton,
New Jersey, USA

Abstract. Java is widely used on the Internet, which makes it a target of choice for malicious attackers. This fact stimulates the research work in the field of Java program verification in order to consolidate both Java safety and security. The results achieved so far in this sector are very promising and effective. Nevertheless, the current Java security architecture still suffers from some weaknesses in terms of flexibility, efficiency and robustness. We therefore propose an architecture, named Java Certifying Compilation (JACC) system, for secure compilation and execution of Java mobile code. This architecture is based on a synergy between certifying compilation and formal automatic verification à la model-checking. Indeed, we have extended the certifying compilation and model-checking approaches to enforce high-level security policies in Java. In this paper, we present our work in designing and implementing the JACC system, which includes a certifying compiler, a security policy specification language and an extended bytecode verifier that integrates a model-checker. This system is flexible, efficient and robust and can be used to enforce both the safety and the security of Java programs.

1 Introduction

Information technology is becoming, more and more, a vitally important underpinning to our economy and to our society. It is embedded in our everyday applications and animates a wide class of systems that range from small to large and from simple to extremely sophisticated. Actually, information systems increasingly govern nearly every aspect of our lives. This omnipresence is increased by the dazzling expansion of Internet, World Wide Web, Java, parallel and distributed systems and mobile computation.

Lately a surge of interest has been expressed in mobile code technology. The latter stands for systems in which general-purpose executable contents can run in remote locations. The idea of mobile code is not new, but it becomes an invaluable cutting edge technology in the presence of World Wide Web and Java compiled units, i.e., the so-called applets. This combination becomes a synergy that caters for an easy, natural and flexible development and distribution of intranet/internet, concurrent and distributed applications. Accordingly, plenty

of systems have been advanced for creating and using mobile code. The most prominent systems are Java, JavaScript, VBScript and ActiveX.

The Java language [1–3] emerged as a multi-paradigmatic language that supports mobile code. It obeys to the paradigm "write once, run everywhere". Indeed, programs in a compiled platform-independent form (class file or Java bytecode) can migrateover the network. A Java bytecode can be executed on any platform that is endowed with a Java virtual machine (JVM) that emulates a processor architecture. The most popular way to achieve code mobility in Java is by putting links in web pages to Java class files (usually referred to as applets).

Mobile code in general and Java in particular poses severe, and very interesting, challenges in terms of security, reliability and performance. The security issue is of paramount importance. The host client accepting a mobile code must check whether the latter will not affect the secrecy (by leaking sensitive information), the integrity (by corrupting information), the authentication (by impersonating authorized principals), the availability (by denying service to legal users), etc. The current trend in mobile code and Java security is defensive (adding layers of firewalling, cryptographic protocols, network partitions, etc.), restrictive (sandbox models, rigid security policies, etc.) and ad hoc (dynamic checks, checksums, scanning, etc.).

Consequently, there is a desiderata that consists in elaborating a security architecture that is flexible, application-dependent, efficient and practical. Moreover such an architecture must be based on robust theoretical foundations.

The Java language uses multi-level mechanisms [4] in order to ensure its protection. More precisely, those mechnaisms are embedded at the language, the compiler, the bytecode verifier and the security manager levels. This architecture is one of the best to ensure safe and secure execution of Java applications. However, it lacks a few interesting properties:

- Flexibility: the present security architecture of the JVM is rigid because it is application-independent. Security policies are only a series of access permissions for resource usage independently of the execution context.
- Efficiency: security property verification is performed exclusively dynamically by the JVM.
- Robustness: the security architecture of the JVM is vaguely and sparsely specified in the official documentation published by JavaSoft Inc. The innerworkings of the bytecode verifier are not well documented. Moreover, numerous errors have been found in the bytecode verifier which shows that its development has not been done by following robust theoretical foundations.

In our current research, we are concerned with the formal static verification of expressive security properties written for Java programs in order to enhance the current Java Security Architecture. To address this problem, we have studied several approaches used to ensure safe local execution of untrusted code. Among them, certifying compilation seems to be a very promising approach, which is based on programming language theory and implementation. The main idea of certifying compilation is to generate a certified code that can be verified by the

consumer using a *verifier* to check if the software complies to a pre-established safety policy.

It is with this strategy in mind that we have developed this research project, named Java Certifying Compilation (JACC) system. We have elaborated an architecture for secure compilation and execution of Java mobile code that is based on an extension of the certifying compilation approach. This architecture solves the three main problems previously enumerated. We have therefore designed and implemented a Java certifying compiler that inserts new type annotations into Java class files, a language for the specification of expressive security policies and a bytecode verifier that integrates a model-checker.

The rest of the paper is organized as follows. We begin with a description of the related work (Section 2). We then present our approach (Section 3). In Section 4, we introduce JACC. We continue with the description of the security policy specification language (Section 5) where we focus on its syntax and semantics. Subsequently, in Section 6, we describe the JACC bytecode verifier, including the bytecode/JACC annotations correspondence verification (Section 6.1) and the JACC model-checker (Section 6.2). Section 7 proposes a case study. Finally, a few concluding remarks and a discussion of future research work are ultimately sketched as a conclusion in Section 8.

2 Related Work

To meet the growing need in Information Technology Security, several approaches were proposed to ensure secure local execution of applications. Many of these are based on dynamic and static analyses. The latter, generally using only the source code or object code of an application, prove very effective in many cases and save execution time compared to dynamic approaches. Nevertheless, most of the static analysis methods are usually difficult to develop. This makes the approaches based on certification interesting. The principle of certifying compilation consists in generating from a source code, by the means of a certifying compiler, an object code that incorporates a certificate of safety or security. This certificate makes it possible for a user of the object code to establish, by means of an object code verifier, the conformity of the object code with a security policy, implicitly or explicitly specified. Contrarily to the traditional approach, where code safety verification falls within the competence of the consumer, this task is distributed between the producer and the consumer.

Self-certified code approaches can be structured in three different classes: the PCC (Proof-Carrying Code), the TAL (Typed Assembly Language) and the ECC (Efficient Code Certification) models.

In the PCC model [5, 6], the consumer requires from the producer a proof of safety (certificate) that attests that the received code complies with definitive and published safety rules. Afterwards, the task of the consumer consists in validating this proof, thereby ensuring that the received code can be safely executed.

The certificate in the TAL model [7, 8] constitutes a type annotation that contains a static approximation of the dynamic behavior of the program. Before any execution of the object code, a verification process is activated to check the conformity of the assembly code compared to the integrated type annotations. This verification is done using a verifier that tests the safety of the execution stack, the control flow and typing. Once the code has been checked, the annotations can be removed and the code can be safely executed.

The ECC model [9] sacrifices expressivity and generality to the profit of performance. This is why ECC is mainly focused on fundamental safety aspects (control flow, memory accesses and execution stack management). This approach includes two principal components: a certifying compiler and a verifier. The first generates an object code and a safety certificate. This certificate corresponds to a set of structured annotations. The verifier uses these annotations to ensure the veracity of a set of conditions.

In the following, we examine the state of the art in Java program verification.

Klaus Havelund and his coauthors have worked on the elaboration of a framework for the verification and debugging of Java programs [10]. This framework is based on model-checking. Their system is specifically designed to verify multi-threaded programs. Their work consists in the definition of Java PathFinder (JPF), a Java to PROMELA (PROcess MEta LAnguage) translator. PROMELA is the input language of the model-checker Spin. This work is interesting but has a few limitations. Indeed, it is not easily extensible to large industrial applications. Furthermore, more work should be done in order to translate all aspects of Java to PROMELA. This is a hard task, because Java and PROMELA are totally different and the latter does not necessarily support everything that the former does.

Cormac Flanagan and Stephen N. Freund present a concrete example of annotations that make the detection of thread synchronization errors in Java programs possible [11]. The programmer must manually add these annotations in the Java source code. An adapted Java compiler can then verify that there are no concurrency errors (also named race conditions) during the compilation process. The authors do not consider the whole Java language. Indeed, they have isolated a subset of the Java language that is sufficient to handle concurrent Java programs.

Christopher Colby, Peter Lee, George C. Necula and their coauthors explore the idea of developing a certifying compiler for Java [12]. They concentrate their efforts on type safety only, which is a safety property and not a security property.

Eran Yahav presents a new verification technique based on parametric frameworks where it is possible to verify given safety properties on concurrent Java programs [13]. The operational semantics of instructions and conditional expressions of the Java language is specified using a meta-language based on first-order logic. The same meta-language is also used to express safety properties. The general idea of this technique is to generate multiple program configurations from an original configuration. These configurations are calculated by applying the semantics of the instructions and expressions of the Java language. It is finally

verified that all these reachable configurations satisfy the given safety properties. The safety properties that are considered in this system are deadlocks, interference, shared abstract data types and illegal thread interactions.

3 JACC Approach

In the JACC architecture, the Java language stays totally unchanged. Therefore, the Java source code that is compiled is the exact same source code that Java developers are used to write. However, the Java compiler is modified. Our certifying compiler is called JACC. Instead of producing only the bytecode and the standard type annotations, it also produces new annotations called JACC annotations. Instead of being directly executed by the Java Virtual Machine, the augmented class files are verified by a modified Java bytecode verifier called JACC bytecode verifier. This verifier guarantees both safety properties and high-level security policies. The safety properties stay the same but the security policies are now expressed with a custom and more expressive language based on the modal μ-calculus. These new security policies are called JACC Security Policies. Once the bytecode is successfully verified, it can be executed by a traditional Java Virtual Machine. Since high-level security is already guaranteed, the Java Security Manager can be turned off and efficiency is therefore increased. Finally, secure execution of the bytecode is also achieved.

JACC and its annotations, security policies and bytecode verifier are presented in the following sections.

4 JACC Certifying Compiler

JACC is a prototype of a certifying compiler for Java source code. It generates, in addition to the bytecode, new annotations (*certificate*) that deal with security concerns. This *certificate* is statically verified by the JACC bytecode verifier. The JACC certifying compiler was developed by extending an existing Java compiler called Jikes (IBM) [14].

4.1 JACC Annotations

The annotations try to capture the behavior of a piece of software and represent it in an abstract form. Presently, five main categories of annotations are considered: file annotations describing file manipulations, system call annotations to bring out calls to the system, network access annotations (URL, socket, datagram), thread annotations describing thread manipulations and window annotations describing window manipulations.

JACC generates annotations for each opcode of a method. This helps to verify that the annotations and the bytecode correspond and that none has been altered when the user receives the classes (see Section 6.1). Table 1 gives the syntax of the annotations.

Table 1. Annotations Syntax

Γ	$::=$	$\Gamma;\Gamma' \mid rec(\Gamma,\Gamma') \mid if(\Gamma,\Gamma') \mid if(\Gamma,\Gamma',\Gamma'') \mid try(\Gamma,\Omega) \mid$
		$try(\Gamma,\Omega,finally(\Gamma')) \mid switch(\Gamma,\Phi) \mid switch(\Gamma,\Phi,default(\Gamma')) \mid \#n(\Lambda)$
Ω	$::=$	$catch(\Gamma) \mid \Omega, catch(\Gamma)$
Φ	$::=$	$case(\Gamma) \mid \Phi, case(\Gamma)$
Λ	$::=$	$\emptyset \mid or(\Lambda,\Lambda') \mid \alpha \mid \beta \mid \varphi \mid \kappa \mid \omega \mid \sigma \mid \lambda \mid \theta$
α	$::=$	$\emptyset \mid value(x,t) \mid variable(\_V_i,t)$
φ	$::=$	$\emptyset \mid open(x,F) \mid read(x,F) \mid write((x_1,k),(x_2,F)) \mid delete(x,F)$
κ	$::=$	$\emptyset \mid open(x,C) \mid read(x,C) \mid write((x_1,k),(x_2,C)) \mid delete(x,C)$
ω	$::=$	$\emptyset \mid open(x,W) \mid read(x,W) \mid write((x_1,k),(x_2,W))$
σ	$::=$	$\emptyset \mid system\_call$
λ	$::=$	$\emptyset \mid native\_method\_call$
θ	$::=$	$\emptyset \mid create(\_V_i) \mid current(\_V_i) \mid start(\_V_i) \mid stop(\_V_i) \mid join(\_V_i) \mid$
		$destroy(\_V_i) \mid interrupt(\_V_i) \mid sleep(\_V_i) \mid suspend(\_V_i) \mid resume(\_V_i)$

In this syntax, Γ corresponds to the annotations of a method, and Λ corresponds to those of a single opcode.

The term α represents the memory effects, φ the file effects, κ the communication effects, ω the window effects, σ the system calls, λ the calls to native methods and θ the thread effects. The separator ; delimits the annotations of different opcodes. The expression $rec(\Gamma,\Gamma')$ represents a loop statement where Γ corresponds to the annotations of the loop condition and Γ' corresponds to the annotations of the loop body. The expressions $if(\Gamma,\Gamma')$ and $if(\Gamma,\Gamma',\Gamma'')$ are used to represent an *if* statement where Γ corresponds to the annotations of the condition, Γ' corresponds to the annotations of the *then* branch and Γ'' corresponds to the *else* branch. The expressions $try(\Gamma,\Omega)$ and $try(\Gamma,\Omega,finally(\Gamma'))$ represent a *try* ... *catch* ... *finally* statement. The *try* block annotations are represented by Γ, Ω corresponds to the annotations representing the list of *catch* blocks and $finally(\Gamma)$ corresponds to the annotations representing the *finally* block. The expressions $switch(\Gamma,\Phi)$ and $switch(\Gamma,\Phi,default(\Gamma'))$ represent the annotations of a *switch* statement, with Γ being the annotations of the condition, Φ the annotations of the different *case*s and $default(\Gamma')$ the annotations of the *default* block. The annotation $\#n(\Lambda)$ is used to assign an opcode offset (n) to the corresponding opcode annotation (Λ).

The annotation $or(\Lambda,\Lambda')$ represents an alternative between two annotations Λ and Λ'. It occurs when the same variable has been assigned to two different values in an *if* statement, for example.

Let us now see the atomic annotations. The first category of atomic annotations capture memory effects: $value(x,t)$ corresponds to an opcode that declares a literal, x is the value of the literal and t is its type. The annotation $variable(\_V_i,t)$ corresponds to an uninitialized field, a method parameter or a result returned by a method or a constructor that we do not consider critical. $\_V_i$ is the variable name (where i is the variable number) and t is its type.

The file effects capture file openings, deletions, and readings from or writings to a file. The network effects capture connection openings, sendings or receivings. The window effects are useful especially when the software tries to get confidential information by simulating interfaces generally used by the operating system. For these three categories we use the same annotations: $open(x, K)$ for opening, $read(x, K)$ for reading/receiving, $write((x_1, k), (x_2, K))$ for writing/sending and $delete(x, K)$ for deleting (files and connections only). In these annotations, K is F for files, C for connections and W for windows, and x, x_1 and x_2 represent the file name, the connection or the window number. The couple (x_1, k) in *write* represents the information to write. This information could be read from a file, a connection or a window, and thus (x_1, k) is identical to the parameters (x, K) of the *read* annotation.

The annotations *system_call* and *native_method_call* are respectively used to capture calls to the system and to native methods, which could be malicious, depending on the actions of these methods.

Threads could be used to introduce attacks such as denial of service (DoS). Thread annotations capture thread manipulations to detect such attacks. In these annotations, $\_V_i$ is a name assigned to a thread to distinguish between different threads.

According to the effects we are trying to bring out (file manipulations, network accesses, thread manipulations, ...), it is important to point out that not all the opcodes are considered relevant. Therefore, these irrelevant opcodes are being attributed *nil* annotations ($\emptyset$).

4.2 Critical Methods

For opcodes dealing with method or constructor invocations, the annotations depend on whether the method or the constructor is critical or not. A method or a constructor is considered critical if it is used to manipulate files, connections, threads, windows or system calls. Indeed, such manipulations could be malicious and affect the secrecy, integrity, etc, of the system. For instance, methods for opening, reading from or writing to files are considered critical since they could manipulate confidential files. We have collected a non-exhaustive set of critical methods from the Java API. The rational underlying that choice is that security breaches at the application level could not be achieved unless some resourse is accessed. The resource could be a file, a thread, a window, the network, the screen, etc. In the case of a non-critical constructor or method that does not return *void*, we generate the annotation $variable(\_V_i, t)$ where $\_V_i$ is a new variable name (i is a number incremented each time a new variable is created). If a non-critical method returns *void*, we generate a *nil* annotation.

Any expression inside a method could use class fields or local variables. To generate the appropriate annotations for these expressions we use an environment that helps us keep track of the annotations of program variables and propagate them when needed. For example, if we have the statement `f.delete()`, and in the environment the annotation $open(name, F)$ is bound to the variable `f`, then we are able to generate the annotation $delete(name, F)$ for this statement.

5 JACC Security Policies Specification Languages

In this section, we introduce two languages that are helpful for defining security policies. The first one, the rather *low-level* Security Policies Specification Language, is based on the modal μ-calculus [15]. It is internally used by the certifying compilation system that we have developed. The second one, the *high-level* Security Policies Specification Language, is used by the end-user of the system to define security policies. Those security policies are properties that enhance the current Java Security Architecture [16]. Indeed, they are new kinds of properties that must be satisfied by the programs in order for the Java platform to execute them. For example, a security policy could specify that some confidential file can be read by some principal but cannot be sent over the network. Such a restriction cannot be specified using the current Java Security Architecture.

It is therefore of paramount importance to tackle this problem if we want to prevent fancy attacks. The Java certifying compilation system that we are developing, and particularly the security policies specification languages, offer a greater flexibility to the end-user, who can now define more complex and complete security policies.

5.1 Low-level Security Policies Specification Language

The low-level Security Policies Specification Language internally used by the certifying compilation system is based on the modal μ-calculus.

The syntax of the low-level Security Policies Specification Language is given in Table 2. This syntax is based on the modal μ-calculus syntax [17]. The main extension we have made consists in parameterizing the actions to propagate some data information in the formula. This helped us to achieve a better precision in the verification process. For example, if a file is read and then a file is sent over the network, we can determine if it is the same file or not. This means that it is not necessary to block all the files at the network level to be sure that a given file will not be sent.

Table 2. Syntax of the Low-level Security Policies Specification Language

ϕ	$::=$	$\mathsf{t} \mid \mathsf{f} \mid Z \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid [K]\phi \mid \langle K\rangle\phi \mid \mu Z.\phi \mid \nu Z.\phi$

We now explain the syntax of the low-level Security Policies Specification Language. We assume that a finite set of actions in Act is given (see Table 3). The formulas t and f respectively represent the assertions *true* and *false*. The formula Z represents a variable; there is an infinite countable set of variables. The $\neg$ operator gives the negation of a given formula. The operators $\wedge$ and $\vee$ respectively return the conjunction (the "and") and the disjunction (the "or") of two formulas. The modal operators $[K]$ and $\langle K\rangle$, where K is a finite set of

actions in Act, respectively express necessity and possibility. When $K = \{a\}$, for some a (that is, K contains a single action), we simply write $[a]$ instead of $[\{a\}]$ and $\langle a \rangle$ instead of $\langle \{a\} \rangle$. The operators μ and ν are respectively the least and greatest fixed point operators. They are the most complex but also the most powerful operators. Among other things, they let us represent infinite properties in a finite manner.

Finally, we mention that, as for the μ-calculus, the operators of the language are not independent, since the following relations exist between them:

$$\begin{aligned}
\mathsf{t} &= \nu Z.Z \\
\mathsf{f} &= \mu Z.Z \\
\mu Z.\phi &= \neg \nu Z.\neg \phi[\neg Z/Z] \\
\nu Z.\phi &= \neg \mu Z.\neg \phi[\neg Z/Z] \\
\phi_1 \vee \phi_2 &= \neg(\neg \phi_1 \wedge \neg \phi_2) \\
\phi_1 \wedge \phi_2 &= \neg(\neg \phi_1 \vee \neg \phi_2) \\
\langle K \rangle \phi &= \neg [K] \neg \phi \\
[K] \phi &= \neg \langle K \rangle \neg \phi
\end{aligned}$$

In these expressions, $\phi[\neg Z/Z]$ is the formula obtained by replacing all free occurrences of Z in ϕ by $\neg Z$.

Let us examine the syntax Act of actions. We are interested in actions that use the resources of the system.

Each action in Act abstracts a particular behavior of a Java program, e.g. opening a file, deleting a file and sending information over the network. Act is presented in Table 3. Most actions are parameterized; this is useful to link data to actions that use them.

Table 3. The Syntax of Actions (Act)

Act	::=	$open(n,k) \mid read(n,k) \mid write((n,k),(n',k')) \mid delete(n,k) \mid$ $native\_method\_call \mid system\_call \mid create(v) \mid current(v) \mid$ $start(v) \mid stop(v) \mid join(v) \mid destroy(v) \mid resume(v) \mid sleep(v) \mid$ $suspend(v) \mid interrupt(v)$

The symbol n designates a file, a window, a connection or an unknown resource. It can be a constant or a variable that will be initialized during the verification process. The use of variables in actions abstracts the internal program behavior to the user. When the resource is unknown, n must be a variable. The symbol k is used to identify the resource and can take the following values: "file" for file, "connection" for connection, "window" for window and "other" for an unknown resource. The $open(n,k)$ action, for example, represents the opening of a resource n that can be a string constant or a variable representing this resource. The latter can be a file, a window, a connection or an unknown resource. The $write((n,k),(n',k'))$ action corresponds to writing data issuing

from a file, a window, a connection or an unknown resource, abstracted by n, into a file, a window, a connection or an unknown resource, abstracted by n'. The *system_call* action is the abstraction of a system call. The *native_method_call* action is the abstraction of a native method call. The remaining actions represent thread actions. For example, the $create(n)$ action is the creation of a new thread represented by n, which must be a variable.

We now give an example of how to use the low-level Security Policies Specification Language presented in Table 2.

Consider the following formula ϕ_1, that expresses the fact that after a file has been read, its content cannot, immediately or later on, be written into a window:

$$\phi_1 = \nu Z.([read(v, \text{file})](\nu Z.([write((v, \text{file}), (v', \text{window}))]\mathsf{f} \wedge [-]Z)) \wedge [-]Z)$$

where $[-]$ denotes the use of the modality with arbitrary action. The formula stipulates that each time that the action $read(v, \text{file})$, representing the reading of any file abstracted by the variable v, is executed, the action representing the writing of a window, $write((v, \text{file}), (v', \text{window}))$, cannot be executed, nor immediately, neither later on after the execution of other actions.

5.2 High-level Security Policies Specification Language

The low-level security policies specification language is very useful for specifying system properties and expressing the security policies internally used by our certifying compilation system. But, on the other hand, its complexity makes it difficult to use. That was the reason why we proposed a simple, user-friendly, easy to learn and high-level language, which hides the technical details that the end user would need to master in order to define security policies. To achieve this objective, we introduced many interesting macros that abstract the particularities, and also the difficulties, of the low-level security policies specification language. These macros express basic properties that can be used to easily define sophisticated security policies. The syntax of the high-level security policies specification language is presented in Table 4.

Table 4. Syntax of the High-level Security Policies Specification Language

P	$::=$	$true \mid false \mid not(P) \mid and(P_1, P_2) \mid or(P_1, P_2) \mid always(P) \mid$ $eventually(P) \mid never(P) \mid loop(K) \mid implies(P_1, P_2) \mid$ $possible(K_h, P) \mid necessarily(K_h, P)$

The symbol P represents a security property and the term K_h is used to represent a finite set of actions in Act_h (see Table 5). The statement $not(P)$ stipulates that the property P must not be satisfied. The statement $and(P_1, P_2)$ means that both properties P_1 and P_2 must be satisfied. The statement $always(P)$

means that the property P must be satisfied from all the program points reachable from the current one. The statement $inevitable(K_h, P)$ stipulates that one of the actions in K_h is immediately inevitable and any program point to which this action may lead satisfies the property P. The statement $loop(K_h)$ means that one of the actions in K_h can be executed in a loop from a program point reachable from the current one.

Let us examine the syntax of actions Act_h used in the high-level language. These actions slightly differ from those introduced in Table 3 in Section 5.1 (Act). To simplify writing security properties, we have modified some actions. Indeed, the sending over the network of some data issuing from a resource n is represented by the action $send((n,k),n')$ instead of considering it as the writing into a file attached to a network connection, which is the case in the Java language.

The syntax of actions Act_h is presented in Table 5. As for Act, the symbol n designates a resource and the symbol k is used to identify the resource.

Table 5. The Syntax of Actions (Act_h)

Act_h	$::=$	$openf(n) \mid readf(n) \mid writef((n,k),n') \mid deletef(n) \mid openc(n) \mid$ $send((n,k),n') \mid receive(n) \mid native\_method\_call \mid system\_call \mid$ $create(v) \mid current(v) \mid start(v) \mid stop(v) \mid join(v) \mid destroy(v) \mid$ $interrupt(v) \mid sleep(v) \mid suspend(v) \mid resume(v) \mid openw(v) \mid$ $readw(v) \mid writew((n,k),v)$

5.3 Semantics

The semantics is defined using a satisfiability relation. The satisfiability rules for the JACC system direct the process of determining if a given model or program satisfies a given formula or security policy. Indeed, for any formula representing a security property, we should be able to determine if a given abstraction of a method *satisfies* that property or not. If all the methods of a program satisfy a set of properties, or a security policy, we say that the program satisfies the policy.

6 JACC Bytecode Verifier

The JACC bytecode verifier integrates into one product a traditional Java bytecode verifier and a model-checker (refer to [18] and [19] for greater details). The Java bytecode verifier is responsible for ensuring basic low-level security properties — also called safety properties — like memory safety, type safety and control flow safety. The model-checker performs high-level security checks, verifying that a given program respects a certain security policy. When someone runs the JACC bytecode verifier, the Java bytecode verifier performs its low-level

checks and then the model-checker verifies that each method of the program respects the current security policy. In fact, the class files generated by the JACC certifying compiler are parsed by a component found in the JACC bytecode verifier: the JACC Class file Parser. Using the JACC annotations, this parser is responsible for producing an abstract representation of the bytecode called a model. Models of Java programs consist of structured annotations that are extracted from these programs. The structure of the annotations is an abstraction of the control flow one. The model is used by the bytecode verifier in conjunction with the model-checker to verify that it satisfies the JACC security policy. This JACC security policy is also parsed by a component found in the JACC bytecode verifier: the JACC security policy parser. This parser contains a translator that is responsible for the translation of the JACC security policy, expressed in the high-level specification language, to a formula, expressed in the low-level specification language. This formula is the one used by the model-checker.

Two steps have been added to the original Java verification process: bytecode/JACC annotations correspondence verification and model-checking. Both steps are rather complex but very interesting. They are explained in the following subsections.

6.1 Bytecode/JACC Annotations Correspondence Verification

The bytecode/JACC annotations correspondence verification is a very important step in the verification process of the JACC bytecode verifier because it ensures that the bytecode included in the JACC class files (augmented Java class files) exactly corresponds to what is expressed by the associated JACC annotations. This is important because the model-checking algorithms, that are executed right after, enforce the security policies by considering only the JACC annotations and not the bytecode. However, what is ultimately executed by the Java Virtual Machine is, of course, the bytecode and not the JACC annotations. Therefore, it must be guaranteed that the critical actions performed in the bytecode are entirely expressed in the associated JACC annotations and also that all critical actions expressed in the JACC annotations are really performed in the bytecode. This property ensures that the JACC annotations constitute a correct abstraction of the program.

The verification step that ensures that the bytecode of a given Java program really corresponds to the associated JACC annotations, and vice versa, is divided into two steps: correspondence verification and a new dataflow analysis.

The correspondence verification substep is very simple. It ensures that each opcode in the bytecode of a Java program has an associated JACC annotation and also that each JACC annotation has an associated opcode. This is performed on the basis of the opcode offsets that are included in the JACC annotations. Therefore, no semantic analysis of the opcodes or the JACC annotations is performed at this stage. This type of analysis is included or inherent to the dataflow analysis that follows this correspondence verification substep.

The new dataflow analysis substep (Algorithm 1) is more complex, and also more interesting. It basically consists of a new dataflow analysis that makes sure

that the semantics of the JACC annotations corresponds to the semantics of the opcodes, and vice versa. This dataflow analysis is in fact similar to the one already used by the traditional Java bytecode verifier to ensure fundamental safety properties.

Algorithm 1: Dataflow Analysis Used by the JACC Bytecode Verifier

```
Initialization
for each opcode o of the current method do
    set the modified flag of o to false;
    set the associated dataflow information of o to unknown;
end
set the modified flag of the first opcode of the current method to true;
set the associated dataflow information of the first opcode of the current method
to the information provided by the current method's signature;

Main loop
while there is an opcode o with a modified flag set to true in the current method
do
    set the modified flag of o to false;
    simulate the execution of o on its associated dataflow information. This
    consists of verifying the semantics of the JACC annotation associated to o.
    If this verification fails, the entire dataflow analysis fails;
    for each successor s of o do
        merge the associated dataflow information of o to the one associated to
        s. The resulting dataflow information is called r. The merge operator
        is the union on sets;
        if r is not equal to the associated dataflow information of s then
            set the associated dataflow information of s to r;
            set the modified flag of s to true;
        end
    end
end

the dataflow analysis is a success;
```

When the opcode o is a *critical constructor* or a *critical method invocation* opcode, the list of critical classes and methods that has been used by the JACC certifying compiler must be considered in order to determine the correct type of JACC annotation that this opcode must be associated with and also to verify the JACC annotations parameters. Therefore, the generated JACC annotation represents the appropriate action performed by the critical constructor or method invocation. When a *non-critical* constructor is invoked, the associated JACC annotation should be a *variable*, which represents the freshly created object. When a *non-critical* method is invoked, the associated JACC annotation should be *nil*

if the method returns `void` because no value is returned and therefore, no annotation must be produced. Finally, the associated JACC annotation should be a *variable* if the method does not return `void` in order to represent the returned value.

6.2 JACC Model-Checker

The model-checker performs high-level security checks that are carried out by a model-checking algorithm. The latter determines whether or not a transition system $M = (S, C, W, R, Sub)$ is a model for a JACC security policy p'_0. The transition system M is a quintuple, where:

- S is the set of states of the model. Where a state represents a control point of the program's abstract model;
- C is the set of constants c_i of the model;
- W is the set of variables w_i of the model;
- R is a relation that associates a set of state couples (s, t) to an action a such that an edge labeled by a exists between s and t; and
- Sub is a set, initially empty, tracking the substituted constants in the model.

The main method (`eval`) algorithm (Algorithm 2) represents a recursive function which calculates the set of states that satisfy a given security policy. Since the parameters of actions can be variables or constants, the function `eval` computes a set for each action variant.

The algorithm follows these steps:

1. Convert the propery p_0 to its equivalent PNF (positive normal form) p'_0. We say that a formula is in a positive normal form if negations are applied only to the atomic propositions.
2. Compute the set of states at which p'_0 hold ($S' = \texttt{eval}(p'_0)$).

Based on the set of states that satisfy the formula, the security policy verifier gives binary answers (true/false) to the question of formula satisfaction. If the verifier is not able to determine whether the formula is satisfied or not because of dynamic aspects (like manually entered filenames, for example), it reacts conservatively, stating that the formula is not satisfied. In the future, we plan to add more verbose and accurate outputs.

7 Case Study

In this section, an example that exercises the main components of the JACC system is introduced and studied.

Example of a Java Program

```
import java.net.*;
import java.io.*;
```

Algorithm 2: The `eval` Function

input : A transition system representing the model $M = (S, C, W, R, Sub)$, and a security policy p_0 containing the variables $X_1, X_2, \ldots, X_n$.
output : A set containing the states that satisfy p_0.

Function `eval`(M, p_0') **variable** S, S';
begin
  switch *the form of* p_0' **do**
    case *true*
      $S' = S$;
    case *false*
      $S' = \emptyset$;
    // Each recursion variable Y_i corresponds to a set S_i
    case Y_i
      $S' = S_i$;
    case $\neg p$
      $S' = S \backslash$`eval`(M, p);
    case $p \wedge q$
      $S' =$ `eval`$(M, p) \cap$ `eval`(M, q);
    case $p \vee q$
      $S' =$ `eval`$(M, p) \cup$ `eval`(M, q);
    case $\langle delete(n, k)\rangle p$ ***or*** $[delete(n, k)]p$
      $S' =$ `evalDelete`$(M, p, (n, k))$;
    case $\langle open(n, k)\rangle p$ ***or*** $[open(n, k)]p$
      $S' =$ `evalOpen`$(M, p, (n, k))$;
    case $\langle read(n, k)\rangle p$ ***or*** $[read(n, k)]p$
      $S' =$ `evalPossibleRead`$(M, p, (n, k))$;
    case $\langle write((n, k), (n', k'))\rangle p$ ***or*** $[write((n, k), (n', k'))]p$
      $S' =$ `evalWrite`$(M, p, (n, k), (n', k'))$;
    // Here A can be any thread action.
    case $\langle A(v)\rangle p$ ***or*** $[A(v)]p$
      $S' =$ `evalA`(M, p, v);
    case $\langle native\_method\_call\rangle p$ ***or*** $[native\_method\_call]p$
      $S' =$ `evalNativeMethodCall`(M, p);
    case $\langle system\_call\rangle p$ ***or*** $[system\_call]p$
      $S' =$ `evalSystemCall`(M, p);
    case $\langle nil\rangle p$ ***or*** $[nil]p$
      $S' =$ `evalNil`(M, p);
  return (S');
end

```
class TestURL {
    public static void main(String[] args)
        throws Exception
    {
        URL url = new URL("http://somehost/cgi-bin/somecgi");
        URLConnection uc = url.openConnection();
        uc.setDoOutput(true);
        FileReader f = new FileReader("confidential.txt");
        PrintWriter out = new PrintWriter(uc.getOutputStream());
        int c;
        while ((c = f.read()) != -1)
        {
             out.print(c);
        }
        out.close();
        f.close();
    }
}
```

This small Java program opens a network connection to a distant computer and sends to this computer the contents of a file known to be confidential.

7.1 JACC Annotations

When we compile the source code of this example, we get the annotations shown in Table 6 for the method `main`.

The annotation #65537 represents the parameters of the method. It is needed to verify the correspondence. In this case, we have only one parameter which is `args`. The annotation #6 corresponds to the `new URL("http://somehost/cgi-bin/somecgi")` statement. We suppose that a connection is opened when created. This annotation is bound to the `url` variable and propagated to the opcode 11 that represents the invocation of the `openConnection()` method.

The annotation #26 represents the file opening which corresponds to the `new FileReader("confidential.txt")` statement.

To write into the connection, an output stream must be created. This is done by invoking the `getOutputStream()` method from the URLConnection instance (`uc`). The annotation corresponding to this invocation is #35.

Reading `confidential.txt` and sending it are done within a loop (`while`). The `while` statement is represented by the *rec* annotation in which the condition annotations range from opcode 53 to opcode 61 and the loop body annotations range from opcode 46 to opcode 50.

Reading the `confidential.txt` file in the loop condition (`f.read()`) is represented by the annotation #54. Sending the character read via the connection is actually writing it to the instance `PrintWriter`. This is represented by the annotation #50 where we can see the source and the destination of the information.

Table 6. Annotations of the Method `main`

$\#65537(params(param(variable(\_V0, \text{“}[Ljava/lang/String;\text{”}))));\#0();\#3();$
$\#4(value\text{“}\texttt{http://somehost/cgi-bin/somecgi}\text{”}, \text{“}Ljava/lang/String;\text{”}));$
$\#6(open(\text{“}\texttt{http://somehost/cgi-bin/somecgi}\text{”}, C));\#9();$
$\#10(open(\text{“}\texttt{http://somehost/cgi-bin/somecgi}\text{”}, C));$
$\#11(open(\text{“}\texttt{http://somehost/cgi-bin/somecgi}\text{”}, C));\#14();$
$\#15(open(\text{“}\texttt{http://somehost/cgi-bin/somecgi}\text{”}, C));\#16(value(\text{“}1\text{”}, \text{“}Z\text{”}));$
$\#17();$
$\#20();\#23();\#24(value(\text{“}confidential.txt\text{”}, \text{“}Ljava/lang/String;\text{”}));$
$\#26(open(\text{“}confidential.txt\text{”}, F));\#29();\#30();\#33();$
$\#34(open(\text{“}\texttt{http://somehost/cgi-bin/somecgi}\text{”}, C));$
$\#35(open(\text{“}\texttt{http://somehost/cgi-bin/somecgi}\text{”}, C));$
$\#38(open(\text{“}\texttt{http://somehost/cgi-bin/somecgi}\text{”}, C));\#41();\#43();$
$rec(\#53(open(\text{“}confidential.txt\text{”}, F));\#54(read(\text{“}confidential.txt\text{”}, F));\#57();$
$\#58();\#60(value(\text{“}-1\text{”}, \text{“}I\text{”}));\#61(),$
$\#46(open(\text{“}\texttt{http://somehost/cgi-bin/somecgi}\text{”}, C));$
$\#48(read(\text{“}confidential.txt\text{”}, F));$
$\#50(write((\text{“}confidential.txt\text{”}, F), (\text{“}\texttt{http://somehost/cgi-bin/somecgi}\text{”}, C))));$
$\#64(open(\text{“}\texttt{http://somehost/cgi-bin/somecgi}\text{”}, C));\#66();$
$\#69(open(\text{“}confidential.txt\text{”}, F));\#70();\#73()$

7.2 JACC Security Policy

In this section, we explain two security policies corresponding to the given example.

Security Policy 1 Consider a property corresponding to the method `main` of the program:

$$never(possible(send((\text{“}\texttt{confidential.txt}\text{”}, \texttt{file}), v), true))$$

This property stipulates that the file named `confidential.txt` can never be sent over the network. After analyzing the annotations of the method `main`, the JACC model-checker states that the program does not satisfy the aforementioned property.

Security Policy 2 Here is another property introduced to verify the method `main` of the program:

$$always(necessarily(readf(\text{“}\texttt{confidential.txt}\text{”}),$$
$$never(possible(send((\text{“}\texttt{confidential.txt}\text{”}, \texttt{file}), v), true))))$$

This property expresses the fact that once the file `confidential.txt` has been read, it cannot immediately or later on be sent over the network. This property is not satisfied by the program, since the file `confidential.txt` is read inside a while statement and then copied into the stream bound to an open connection.

8 Conclusion

In this paper, we have presented our work on defining and implementing an expressive architecture, named Java Certifying Compilation (JACC) system, for secure compilation and execution of Java mobile code based on certifying compilation. The purpose of the JACC system is to statically detect as much potential malicious behaviors as possible in Java programs. In fact, it is a language-based approach to handle security problems.

The JACC system includes a certifying compiler and a bytecode verifier. The JACC certifying compiler generates annotations in addition to the bytecode. These annotations try to capture every critical program behavior. The JACC bytecode verifier analyzes the generated annotations using a model-checker and verifies if they respect a certain security policy. The JACC system enables us to statically detect several cases of suspicious program behavior. The most useful feature of the security policy specification language is the possibility to use parameterized actions. Thanks to these actions, we are able to find the file names and URLs used by a program. In some cases, where data is not statically known, the use of variables as action parameters has shown to be very useful as an abstraction for missing data. This means that even if we cannot certify with certainty that a particular file is used, we can at least suspect it.

The JACC system is more flexible (the security policy language is expressive and permits to specify security properties), more efficient (the verification process is exclusively static) and more robust (the security model is based on formal foundations) than the existing Java security architecture.

A major challenge is to extend our JACC system to handle the following features:

- Perform interprocedural analyses to detect attacks that include cooperation between several methods.
- Add more verbose and accurate outputs to the security policy verifier.
- Extend the range of possible values of the variable k, used in actions, in order to include other resources such as the screen, the keyboard, etc.

References

1. James Gosling, Bill Joy, G.S., Bracha, G.: The Java Language Specification. Second edn. The Java Series. Addison Wesley (2000) `http://java.sun.com/docs/books/jls/index.html`
2. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Second edn. The Java Series. Addison Wesley (1999) `http://java.sun.com/docs/books/vmspec/index.html`
3. Mary Campione, K.W., Huml, A.: The Java Tutorial. Third edn. The Java Series. Addison Wesley (2001) `http://java.sun.com/docs/books/tutorial/index.html`
4. Gong, L.: Java Security Architecture (JDK 1.2) (1998) `http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-%spec.doc.html`

5. Necula, G.C.: Proof-Carrying Code. In: Proceedings of the Twenty-Fourth ACM Symposium on Principles of Programming Languages, Paris, France (1997) 106–119
6. Necula, G.C., Lee, P.: Safe, Untrusted Agents Using Proof-Carrying Code. In Vigna, G., ed.: Mobile Agents and Security. Springer-Verlag LNCS 1419 (1998) 61–91
7. Morrisett, G., Walker, D., Crary, K., Glew, N.: From System F to Typed Assembly Language. In: Twenty-Fifth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA (1998) 85–97
8. Morrisett, G., Crary, K., Glew, N., Grossman, D., Samuels, R., Smith, F., Walker, D., Weirich, S., Zdancewic, S.: Talx86: A Realistic Typed Assembly Language. In: In ACM SIGPLAN Workshop on Compiler Support for System Software, Atlanta, GA, USA (1999) 25–35
9. Kozen, D.: Efficient Code Certification. Technical Report TR98-1661, Cornell University (1998)
10. Havelund, K., Pressburger, T.: Model Checking Java Programs Using Java Pathfinder. Software Tools for Technology Transfer **2** (1999) 366–381
11. Flanagan, C., Freund, S.N.: Type-Based Race Detection for Java. Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation **35** (2000) 219–232
12. Colby, C., Lee, P., Necula, G.C., Blau, F., Plesko, M., Cline, K.: A Certifying Compiler for Java. In: Proceedings of the Conference on Programming Language Design and Implementation, Vancouver, Canada, ACM Press (2000) 95–107 `http://www.cs.berkeley.edu/~necula/pldi00b.ps.gz`
13. Yahav, E.: Verifying Safety Properties of Concurrent Java Programs Using 3-Valued Logic. Proceedings of the Twenty-Eighth ACM SIGPLAN-SIGACT symposium on Principles of programming languages **28** (2001) 27–40
14. IBM: Jikes Official Homepage. `http://oss.software.ibm.com/developerworks/opensource/jikes/` (2002)
15. Stirling, C.: Modal and Temporal Logics for Processes. Technical Report ECS-LFCS-92-221, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh (1992)
16. JavaSoft: Security (1998) `http://java.sun.com/products/jdk/1.2/docs/guide/security/index.html`
17. Kozen, D.: Results on the Propositional μ-calculus. Theoretical Computer Science **27** (1983) 333–353
18. Klaus Havelund, J.U.S.: Applying Model Checking in Software Verification. In: Proceedings of the Sixth International SPIN Workshop on Practical Aspects of Model Checking, Toulouse, France (1999) 216–231
19. Park, D.Y.W., Stern, U., Skakkebaek, J.U., Dill, D.L.: Java Model Checking. In: Proceedings of Automated Software Engineering conference, Grenoble, France (2000) 253–256