Defence Research and Development Canada | Recherche et développement pour la défense Canada

DEFENCE R&D DÉFENSE

Java hybrid analysis version 0.5

F. Painchaud
DRDC Valcartier

Defence R&D Canada – Valcartier
Technical Memorandum
DRDC Valcartier TM 2004-060
February 2007

Canada

Java hybrid analysis version 0.5

F. Painchaud
DRDC Valcartier

Defence R & D Canada – Valcartier

Technical Memorandum

DRDC Valcartier TM 2004-060

February 2007

Author

Frédéric Painchaud, M.Sc.

Approved by

Yves van Chestein, M.Sc., ing.f.
Head/Information and Knowledge Management Section

Approved for release by

Christian Carrier
Chief Scientist

Abstract

This document proposes a new program analysis technique, called *hybrid analysis*, that combines both static and dynamic analysis principles. It is currently being developed to ensure the security of Java programs in the context of critical military information systems. This new program verification technique is theoretically sound and precise, which constitutes a major contribution to the formal program verification domain.

At this stage, Java hybrid analysis is based on finite state automata theory and first-order predicate logic, only succinctly covered in this document. The reader is thus assumed to be relatively familiar with these domains. However, the precise model and logic used by Java hybrid analysis are extensively described. Furthermore, five different hybrid analysis approaches have been devised. They have been given the following names: *interactive hybrid analysis*, *parameterized hybrid analysis*, *test-based hybrid analysis*, *worst-case hybrid analysis*, and *statically-supported dynamic analysis*. A small case study is presented in order to illustrate the concepts behind the first approach. Finally, a discussion depicts the future of this approach.

The Java hybrid analysis detailed in this document is labeled version 0.5 to depict the fact that it is still unstable and evolving.

Résumé

Ce document propose une nouvelle technique d'analyse de programmes, appelée *analyse hybride*, qui combine des principes d'analyses statique et dynamique. Elle est présentement développée pour assurer la sécurité de programmes Java dans le contexte des systèmes d'information militaires critiques. Cette nouvelle technique de vérification est théoriquement saine et précise, ce qui constitue une contribution majeure au domaine de la vérification formelle de programmes.

À ce stade, l'analyse hybride pour Java est basée sur la théorie des automates finis et la logique des prédicats, qui sont seulement succinctement présentées dans ce document. Le lecteur est donc supposé connaître relativement bien ces domaines. Cependant, le modèle et la logique précis utilisés par l'analyse hybride pour Java sont décrits en détail. De plus, cinq approches différentes d'analyse hybride ont été imaginées. Elles ont les noms suivants : *analyse hybride interactive*, *analyse hybride paramétrée*, *analyse hybride basée sur les tests*, *analyse hybride en pire cas* et *analyse dynamique supportée par l'analyse statique*. Une petite étude de cas est aussi présentée de façon à illustrer les concepts sous-jacents à cette première approche. Finalement, une discussion envisage le futur de cette approche.

L’analyse hybride pour Java détaillée dans ce document est étiquettée par la mention “version 0.5” pour illustrer le fait qu’elle est encore instable et en évolution.

Executive summary

The Java Architecture is believed to be one of the best commercial architectures with which to build C2IS and other critical military information systems because it includes high-quality mechanisms to enforce security policies such as static verification (i.e., prior to execution) and run-time monitoring (i.e., during execution). However, optimized configurations and specialized extensions are needed to satisfy military requirements in terms of safety, reliability and security. Harmonizing the Java Security concepts with validated design and execution surveillance has been identified as a top R & D priority to allow continuous risk management.

The novel software verification approach, called *hybrid analysis*, that this document presents combines static verification and run-time monitoring principles in order to successfully fulfill these paramount safety-, reliability- and security-related military requirements. Indeed, by integrating advanced mechanisms to enforce security policies into one consolidated analysis and by allowing these mechanisms to communicate and cooperate, hybrid analysis greatly reduces the chances of executing malicious code. The idea of this approach came from the experience of the MaliCOTS Project, that demonstrated the need for a verification technique that unifies static and dynamic analysis principles in a common framework.

F. Painchaud; 2007; Java hybrid analysis version 0.5; DRDC Valcartier TM 2004-060; Defence R & D Canada – Valcartier.

Sommaire

Plusieurs croient que l'architecture Java est une des meilleures architectures commerciales pour développer des systèmes d'information de commandement et contrôle, et autres systèmes d'information critiques militaires, parce qu'elle contient des mécanismes de qualité pour mettre en application des politiques de sécurité, comme la vérification statique (c'est-à-dire avant l'exécution) et le monitorage dynamique (c'est-à-dire pendant l'exécution). Cependant, des configurations améliorées et des extensions spécialisées sont nécessaires pour satisfaire aux exigences militaires en matière de sûreté, de fiabilité et de sécurité. L'harmonisation des concepts de sécurité Java, de conception validée et de surveillance de l'exécution est considérée comme une priorité importante en R & D afin de permettre la gestion continue du risque.

La nouvelle approche de vérification logicielle, appelée *analyse hybride*, présentée dans ce document combine des principes de vérification statique et de monitorage dynamique de façon à mieux satisfaire les besoins militaires en matière de sûreté, de fiabilité et de sécurité logicielles. En effet, en intégrant des mécanismes avancés pour mettre en force des politiques de sécurité dans une analyse consolidée et en permettant à ces mécanismes de communiquer et de coopérer, l'analyse hybride réduit de façon importante les risques d'exécuter du code malicieux. L'idée de cette approche est venue de l'expérience du projet MaliCOTS, qui a démontré le besoin de développer une technique de vérification qui unifie les principes d'analyse statique et dynamique dans un cadre de travail commun.

F. Painchaud; 2007; Java hybrid analysis version 0.5; DRDC Valcartier TM 2004-060; Recherche et développement pour la défense Canada – Valcartier.

Table of contents

1 Introduction

In this document, hybrid analysis is defined as an analysis technique that combines static and dynamic analysis principles such as, for example, traditional static analyses combined with interaction with the analyst (interaction similar to the one used in conventional debugging). Parametrization, discussed later in this document, can also be used in order to reduce the necessity for constant human interaction. This integration of static and dynamic analysis provides more precise analysis results by complementing the information that is statically missing in the program being analyzed with information only available at run-time. Informally stated, hybrid analysis uses a model, encoded using a computational theory, and a property, expressed via a logic or automaton, and determines if the model satisfies the property, that is, if the property is *true* in the model. In this respect, the hybrid analysis described in this document is inspired by model checking, a program analysis technique that takes into account the behavior of analyzed programs.

In this case, hybrid analysis is used to ensure the security of Java programs, expressed at the bytecode level, in the context of critical military information systems. In the current framework, the model is represented by a non-deterministic finite state automaton and the property, by a first-order predicate logic formula. Note that this Java hybrid analysis theory is *not yet* stable and is meant to be *evolving*. This version is developed only to set the stage for using more expressive underlying theories like Abstract State Machines (ASMs) and temporal or modal logics.

Non-deterministic finite state automata theory and first-order predicate logic are described in sections 2 and 3, respectively. Section 4 depicts a few hybrid analysis approaches that could be used in the setting presented in this document. A small case study is also presented in section 5 in order to illustrate the major concepts of the current framework. Finally, section 6 discusses this framework and section 7 presents conclusions.

This research was performed under WBE 15BF34 – “Secure JAVA Exploitation in C2IS”.

2 Model

The model used by Java hybrid analysis is based on traditional non-deterministic finite state automata theory. This theory is very well explained in [1] and therefore, it is not extensively explained in this document. Rather, the model used in the current framework is directly detailed in the following.

Formally stated, a model M is a tuple $(S, \Sigma, \rho, \iota, F)$, where

- S is a finite, non-empty set of states,
- $\Sigma \subseteq I \times P$ is a relation on an alphabet (i.e., a finite, non-empty set of symbols) I and another finite, non-empty set P of parameters,
- $\rho \subseteq S \times \Sigma \times S$ is a transition relation,
- $\iota \in S$ is the initial state, and
- $F \subseteq S$ is the (possibly empty) set of final states.

In other words, a model M is a labelled, directed graph where S is the set of nodes, Σ is the set of labels, ρ is the set of edges (or links), ι is the graph entry point, and F is the graph set of exit (or termination) points. A model is an abstraction of a program. It should contain all the information necessary to reason about the program, without the unnecessary information. It defines what is to be considered *true* for a given program.

Applying this theory to Java programs expressed at the bytecode level, the different components of a model are as follows:

- S contains the program points,
- Σ contains the parameterized bytecode instructions of the program (see remark 2 in subsection 2.1),
- ρ represents the control-flow between these bytecode instructions,
- ι is the program entry point, and
- F contains the program exit (or termination) points.

2.1 Remarks

Before going on to the description of the logic, a few remarks must be made concerning this model representation:

1. Program points lie *between* each bytecode instruction of the program.
2. In Σ, bytecode instructions in I are parameterized by their *parameter* in P.[1] For example, the `invokevirtual` instruction takes the following parameter: a symbolic reference to the constant pool pointing to the name and signature of the method to invoke. Thus, a program could possibly contain the

[1]Here, it is assumed that all bytecode instructions only take *one* parameter. This is false for a few bytecode instructions but it is easy to compact their parameters into a single, conceptual one.

following instruction: `invokevirtual #19`, where `#19` refers to the method `boolean delete()` (see section 5 for a more complete example).

With this definition of bytecode instructions, there can be an infinite number of them, because there are infinitely-many possible parameters.[2] It means that P, and thus Σ, would not be finite, which contradicts the finite state automata theory assumption. However, for any particular Java program, the parameter of any bytecode instruction is statically-defined (refer to section 5 for a concrete illustration of this fact). Moreover, any Java program is finite in length. Therefore, with this definition of a parameter, P and Σ are really finite for any given Java program.

3. ρ is a *relation* and not a function. Therefore, non-deterministic control flow can be easily and naturally modelled.

4. If F is *empty*, the modelled Java program does not terminate. However, if F is *not* empty, it does not mean that the modelled Java program certainly terminates. It only means that it is possible that it terminates.

5. Since the current focus is placed on verifying *security*, the interesting bytecode instructions are the ones calling security-critical methods of both user- and standard API code. A technique to automatically find security-critical methods in user- and standard API code is discussed in subsection 3.1.1.

6. Theoretically speaking, if it is assumed that the analyzed programs always terminate, the set of runs (or execution paths) of M is the set of sequences accepted by M, noted $L(M)$. Under this assumption, $L(M)$ is therefore a regular language.

3 Logic

The logic used by Java hybrid analysis is based on first-order predicate logic. This logic is quite simple. It is very well known and it is explained in many documents (for instance, refer to [2] for a short introduction). It is summarized in the following paragraphs.

Logic is used to represent properties of objects in the world about which it is necessary to reason. The fundamental components (or entities, constants) of logic are

- objects,
- functions (procedures), and

[2] For instance, strings constitute an infinite domain.

- relations (predicates).

Logic is always used to reason about fundamental elements of a given world. These fundamental elements are called *objects*. Objects are the most atomic entities of the world under consideration. For example, in the model of section 2, objects are the system's security-critical resources (files, network connections, etc.). Indeed, the current Java hybrid analysis needs to be able to reason about security, i.e., about the different accesses to security-critical resources that are performed by the Java program. Therefore, security-critical resources need to be the fundamental elements of the world, expressed via its model.

Functions are mathematical entities that take a certain number of objects as parameters and return an object as a result of computations, which are normally based on these passed parameters. Therefore, in logic, *functions* are used to compute objects from other objects. Together with relations (see below), they are used to specify fundamental, or atomic, properties of objects in the logic. These fundamental properties are called *atomic propositions* (see p in the syntax of subsection 3.1). In the model of section 2, functions are considered to be the bytecode instructions that call security-critical methods. Indeed, these bytecode instructions can be viewed as functions taking a security-critical resource as a parameter (the resource to be accessed) and returning a security-critical resource, which is the access (to the resource) itself. In fact, these functions do not return a tangible object, but their side effect (the access to the security-critical resource) can be conceptually viewed as a return value. In this respect, functions of the model of section 2 could be considered as *procedures*. An example of a bytecode instruction that calls a security-critical method is, once again, `invokevirtual #19`, where `#19` refers to the method `boolean delete()`. The corresponding formula in the current logic to specify such call would simply be: delete. The reasons for this correspondence are given by the semantics of the logic (see subsection 3.2; and section 5 for a more complete example).

Security-critical methods are defined to be the ones that access security-critical resources. For instance, a method that deletes a file is a security-critical method since it accesses a security-critical resource: a file. A bytecode instruction that calls this method would be considered a function (or procedure) in the model. Therefore, in this case, *functions are simply bytecode instructions "of interest"*, i.e., bytecode instructions that manipulate security-critical resources. A technique to automatically find security-critical methods in user- and standard API code is discussed in subsection 3.1.1.

Finally, *relations* (also called *predicates*), establish a link between a certain number of objects. For example, the fictional relation

$$\text{before} \subseteq \text{Objects} \times \text{Objects}$$

could represent the fact that some resource is used before another one. Therefore,

$$(\text{open(“file1.txt”)}, \text{delete(“file1.txt”)}) \in \text{before}$$

would represent the fact that the file “file1.txt” is opened before it is deleted. As noted above, together with functions, relations are used to specify the atomic propositions in the logic. *No* relations are currently used in the current framework, but it does not mean that none will be eventually added. Relations add expressivity to the logic, but at the cost of greater complexity.

3.1 Syntax

Formally stated, the syntax of the current logic is as follows:

$$\Phi ::= p(s) \mid \neg\Phi \mid \Phi_1 \vee \Phi_2 \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \rightarrow \Phi_2 \mid \Phi_1 \leftrightarrow \Phi_2 \mid (\Phi) \mid \forall r : R_i.\Phi \mid \exists r : R_i.\Phi,$$

where $p \in AP$, the set of all *atomic propositions* (i.e., the functions and relations (if any) of the logic), $s \in R$, the set of all security-critical resources (which are, as noted before, the objects of the logic), and r is a typed variable over the set R_i of security-critical resources. The sets R and R_i are later defined below.

The set AP contains all the atomic propositions expressible in the logic. In other words, it contains all the basic security-critical actions that can be performed on the security-critical resources. Therefore, the logic can handle many different security-critical resources like files, network connections, processes, windows, etc. Furthermore, it can also handle many different actions on these resources like open, close, create, delete, read, write, connect, listen, show, hide, etc. For the purposes of this document, only files are considered with the actions open and delete, although the theory scales well to more resources and actions.

Formally stated, in the current framework,

$$AP = \{\text{openfile}, \text{deletefile}\}.$$

The definition of R is

$$R = \bigcup_{i \in I} R_i,$$

where I is a set of indices and each R_i is a set of security-critical resources, one i for each type of security-critical resources (files, network connections, etc.). Therefore, because $s \in R$, s can be viewed as any string that represents the name and location of a file, the URL of a network connection, the name of a process, just to cite a few examples. Note that $\lambda \in R$ is always considered to be *true*, where λ is the empty string, representing no resource. Furthermore, from the definition of R and R_i, it can be concluded that the quantifiers $\forall$ and $\exists$ are typed in the logic.

In theory, the sets R_i (and R by definition) should contain, for a particular system (environment), all the possible security-critical resources. In the current context where only files are considered on atomic propositions open and delete, that corresponds to instantiating these atomic propositions for all files of the system, that is, for example, openfile("file1.txt"), openfile("file2.txt"), ..., deletefile("file1.txt"), deletefile("file2.txt"), ... Of course, this cannot be systematically applied in practice. Therefore, in practice, the sets R_i and R do not have to be completely and concretely defined. They are more like conceptual sets, useful in defining the hybrid analysis theory. It is enough to say that they conceptually contain all the possible security-critical resources, for a particular system. These sets are very large but finite, so they match well with the theory.

Another way of viewing the sets R_i and R would be to let the analyst define them in appropriate configuration files. This would give additional flexibility to the framework.

Going back to the definition of the syntax, the informal, or intuitive meaning of the different syntactic constructions are standard and are not detailed in this document. Subsection 3.2, however, gives their formal semantics in the current framework.

Nevertheless, note that the logical operators have the following precedence (from strongest to weakest):

1. $\neg$
2. $\vee, \wedge$
3. $\rightarrow, \leftrightarrow$

The parentheses "()" are provided for the analyst to override this precedence if needed. They can also be used to clarify the properties and make them easier to read for a human by grouping subproperties.

3.1.1 Finding Security-Critical Methods

A simple but effective technique to automatically find security-critical methods in user- and standard API code is to proceed by regular expression searching in the standard Java API source code. Regular expression searching simply gives more expressivity, more power than standard text searching.

In fact, the Java architecture contains a component called the *Security Manager*. This component is responsible for enforcing a local user-defined security policy at run-time. All the methods of the standard API that access system resources perform a call to the Security Manager before accessing the resource itself, in order to verify that the access is permitted. Therefore, by searching for these calls to the Security Manager, it is possible to find all methods that access system resources, which, by definition, are the security-critical methods of the standard API code. Note that finding the security-critical methods via such a technique is equivalent to exhaustively defining the set AP of atomic propositions (see subsection 3.1).

From there, the security-critical methods of user-code are simply the ones that call at least one of the security-critical methods of the standard API code.

This technique has the advantage of being very simple and static. Once the security-critical methods of a certain version of the standard API have been enumerated, they are known once and for all for all user programs executing on this API version. However, this requires trusting the source code of the standard API. In other words, if a call to the Security Manager is missing from the list of security-critical method of the standard API, this security-critical method will also be missing in the list created by this technique.

3.2 Semantics

The semantics of the logic is interpreted on the model defined in section 2. This is easily understandable by recalling that the model defines what is to be considered *true* for a given program. Therefore, it becomes clear that it is necessary to consult the model in order to determine whether a given property expressed in the logic is true or not.

Formally stated, the semantics of the logic follows:

```
$M \models p(s)$              iff  if $s = \lambda$ then
                                      there exists $s_1, s_2 \in S$ and $(q, r) \in \Sigma$
                                         such that $(s_1, (q, r), s_2) \in \rho$ and $p = q$
                                   else if $s \neq \lambda$ then
                                      there exists $s_1, s_2 \in S$ such that $(s_1, (p, s), s_2) \in \rho$ or
                                      if there exists $s_3, s_4 \in S$
                                            such that $(s_3, (p, \lambda), s_4) \in \rho$ then
                                         Hybrid_Analysis()
$M \models \neg\Phi$            iff  not($M \models \Phi$)
$M \models \Phi_1 \vee \Phi_2$       iff  $M \models \Phi_1$ or $M \models \Phi_2$
$M \models \Phi_1 \wedge \Phi_2$       iff  $M \models \Phi_1$ and $M \models \Phi_2$
$M \models \Phi_1 \rightarrow \Phi_2$       iff  $M \models \neg\Phi_1 \vee \Phi_2$
$M \models \Phi_1 \leftrightarrow \Phi_2$       iff  $M \models (\Phi_1 \rightarrow \Phi_2) \wedge (\Phi_2 \rightarrow \Phi_1)$
$M \models (\Phi)$             iff  $M \models \Phi$
$M \models \forall r : R_i.\Phi$     iff  for all $s \in R_i, M \models \Phi[s/r]$
$M \models \exists r : R_i.\Phi$     iff  there exists $s \in R_i$ such that $M \models \Phi[s/r]$
```

The following cases of this semantics are conventional and therefore, they are not detailed in this document: $M \models \neg\Phi, M \models \Phi_1 \vee \Phi_2, M \models \Phi_1 \wedge \Phi_2, M \models \Phi_1 \rightarrow \Phi_2, M \models \Phi_1 \leftrightarrow \Phi_2$, and $M \models (\Phi)$.

The cases $M \models \forall r : R_i.\Phi$ and $M \models \exists r : R_i.\Phi$, however, are particular because the quantified variable r is typed over a domain R_i. Therefore, their semantics reflects this particularity by quantifying over the same domain. The notation $\Phi[s/r]$ represents Φ where every occurrence of r has been replaced by s. In words, M satisfies (models) the formula $\forall r : R_i.\Phi$ if and only if for all $s \in R_i$, M satisfies Φ, where every occurrence of r has been successively replaced by each s. For example, if $R_i = \{r_1, r_2, \ldots, r_n\}$, $M \models \forall r : R_i.\Phi \equiv M \models \Phi[r_1/r]$ and $M \models \Phi[r_2/r]$ and $\ldots$ and $M \models \Phi[r_n/r]$. The case $M \models \exists r : R_i.\Phi$ is similar but over existentiality instead of universality. Note that for both cases, the typed variable r should be present in Φ in at least one of its $p(r)$'s (that is, $p(s)$'s where s is named r).

Because this document is about hybrid analysis, and not conventional static analysis, the case $M \models p(s)$ is very particular. Indeed, it is divided in two fundamental subcases: if $s = \lambda$ and if $s \neq \lambda$. The first case, where $s = \lambda$, is rather simple and standard. In that case, $M \models p(s)$ if and only if there exists $s_1, s_2 \in S$ and $(q, r) \in \Sigma$ such that $(s_1, (q, r), s_2) \in \rho$ and $p = q$. In other words, M satisfies $p(s)$ if and only if there exists a transition $(s_1, (q, r), s_2) \in \rho$ such that $p = q$. This corresponds to the case where, for example, delete("...") is one of the labels in the model and delete is the property to verify.

The second case, where $s \neq \lambda$, is where hybrid analysis can possibly come into action. But first, it must be verified if it is really necessary to use hybrid analysis. Indeed, if, by chance, there exists $s_1, s_2 \in S$ such that $(s_1, (p, s), s_2) \in \rho$ in M, then it is already concluded that $M \models p(s)$ because a transition explicitly labelled by $p(s)$ (that is, $(s_1, (p, s), s_2)$) has been found in M. This corresponds to the example of subsection 5.4.1.

Otherwise, if there are no $s_1, s_2 \in S$ such that $(s_1, (p, s), s_2) \in \rho$ in M, then it must still be verified if there exists $s_3, s_4 \in S$ such that $(s_3, (p, \lambda), s_4) \in \rho$ in M. If this is *not* the case, it can already be concluded that M does *not* satisfy $p(s)$. Indeed, this would mean that a transition labelled by the abstraction of $p(s)$, that is, $p(s)$ where s has been abstracted out (simply removed to form $p(\lambda)$), could not be found in M. Adding that to the fact that it is already known that there is no transition explicitly labelled by $p(s)$ (this conclusion was drawn in the previous paragraph), it is now certain that M does not satisfy $p(s)$. This corresponds to the case where, for example, there is no delete in the labels of the model but delete("...") is present in the property to verify.

However, if it *is* the case that there exists $s_3, s_4 \in S$ such that $(s_3, (p, \lambda), s_4) \in \rho$ in M, hybrid analysis must be used. It is now clear that hybrid analysis is used only when it could help the analysis to provide more precise results, by using some technique to complement missing information in M (a few examples of such techniques are presented in section 4). In other words, *hybrid analysis uses mathematically sound techniques to inform the analyst when static analysis is no longer sufficient; relatively less sound dynamic techniques are then used to complete the analysis thereby providing more precise results to the analyst*. For the sake of comparison, a standard, conservative static analysis would have been defined with the following rule instead of the one actually used:

```
$M \models p(s)$   iff   if $s = \lambda$ then
                             there exists $s_1, s_2 \in S$ and $(q, r) \in \Sigma$
                                 such that $(s_1, (q, r), s_2) \in \rho$ and $p = q$
                         else if $s \neq \lambda$ then
                             there exists $s_1, s_2 \in S$ such that $(s_1, (p, s), s_2) \in \rho$ or
                                 $(s_1, (p, \lambda), s_2) \in \rho$
```

The only difference is in the last part of the rule, where

there exists $s_1, s_2 \in S$ such that $(s_1, (p, s), s_2) \in \rho$ or $(s_1, (p, \lambda), s_2) \in \rho$

appears instead of

```
there exists $s_1, s_2 \in S$ such that $(s_1, (p, s), s_2) \in \rho$ or
if there exists $s_3, s_4 \in S$ such that $(s_3, (p, \lambda), s_4) \in \rho$ then
    Hybrid_Analysis(),
```

but it makes a big difference. Indeed, the problem with the traditional definition is that it is so conservative (imprecise) that, most of the time, it will conclude that M *does* satisfy $p(s)$ even though in reality, M does *not* satisfy $p(s)$. However, this definition is normal and standard. It is due to the fact that there is missing information in M to conclude with certainty that M does not satisfy $p(s)$. Therefore, in order to have a *conservative* (or *sound*) static analysis, it must state that M satisfies $p(s)$. This way of defining static analyses makes conservative but rather imprecise analyses. Hybrid analysis is a way of defining more precise analyses (see section 6 for a discussion on that matter).[3]

In order to complete missing information in models, a few hybrid analysis approaches have already been identified. They are described in the following section.

4 Hybrid Analysis Approaches

Currently, there are a few potential methods for integrating additional information into the model during analysis, which result in different hybrid analysis approaches. They have been given the following names: *interactive*, *parameterized*, *test-based*, and *worst-case hybrid analysis*. For the sake of completeness, a *statically-supported dynamic analysis* has also been hypothesized. The ideas behind all these approaches are presented in the following subsections.

4.1 Interactive Hybrid Analysis

Interactive hybrid analysis aims at posing appropriate questions to the analyst during analysis. These questions focus on getting information that is not statically present in the model. This is the approach investigated in the current framework.

To sum up this approach by a simple example, suppose that a fileopen is modelled but the file name and location are not known at analysis time. The traditional way

[3]In pure algorithmic/automatic-deduction/computational theory, 100%-precise static analyses are impossible in general. It has been indirectly proven by Gödel, in his famous incompleteness theorem. However, when the analyst can intervene, this is no longer true. If the analyst is considered as an oracle and if this oracle is perfect, everything is possible, at least in theory. Of course, in practice, this comes at the price of the time of the analyst passed in front of the analyzer. Hybrid analysis must therefore be carefully defined in order to limit this cost.

of coping with this lack of information in the model is to abstract it by some means, such as using variables, for instance. However, it has been observed, in earlier projects, that this solution rapidly becomes highly complex and very difficult to manage. Moreover, as mentioned in subsection 3.2, it leads to imprecise results in many situations. Therefore, in the presence of temporal logics that consider execution paths (which is not the case in the current framework), interactive hybrid analysis would cope with this situation by asking the analyst to enter the name and location of a file of interest in order to more precisely verify what happens to it afterwards. For example, the analyst could choose a file that he knows to be confidential in order to determine if the information contained in this file is dealt with in a non-secure manner (with respect to a predefined security policy, i.e., a predefined property). The hybrid analyzer could memorize the various choices of the analyst and create a verification script that would be useful for future verifications. In the absence of temporal logics, a clear and detailed situation is presented to the analyst so that he can assess the risks involved and take appropriate decisions.

By "posing appropriate questions to the analyst", it is not meant that questions should be systematically asked each time an action is too abstract (as is currently the case in the framework). In fact, an analysis should be performed to find which abstract actions should be refined as a priority, depending on the benefits of refining them. It is believed, at this stage, that this analysis could be based on traditional dataflow and dependency analysis techniques.

In other words, questions should be asked optimally, in terms of anticipated benefits for the analysis and minimal annoyance for the analyst. This provides an interesting challenge for this approach.

It could also be interesting to find solutions for the following inter-related problems:

1. How can abstract actions be prioritized in order to optimally determine which ones are needed to be refined first with respect to the property to be verified?

2. How can one identify the smallest set of abstract actions for which refinement would lead to the verification of a given property?

3. How could information be added into the model without interrupting or disturbing the on-going analysis process?

4. How can refined models be abstracted with respect to the verified property or the class of this property in order to facilitate the rest of the current verification process?

4.2 Parameterized Hybrid Analysis

Parameterized hybrid analysis targets a more automatic hybrid analysis approach than interactive hybrid analysis. Indeed, in this approach, the analyst is expected to define resource domains (for example, confidential files, prohibited network usage, etc.) that the hybrid analyzer will use to populate missing information in appropriate abstract actions (hence the name parameterized hybrid analysis).

This approach is essentially based on interactive hybrid analysis except that it has a higher degree of independence from the analyst. It could be interesting for analysts with less background in formal program verification or in contexts where the analyst who uses the analyzer is not the same as the one who determines what has to be verified, i.e., the properties and the critical resources. The same problems, identified in subsection 4.1, would thus also require solutions.

4.3 Test-Based Hybrid Analysis

An addition to parameterized hybrid analysis would be *test-based* hybrid analysis. This would require that the analyst produce a set of test cases that would be automatically verified by the analyzer. In this context, resource domains would also be defined (as for parameterized hybrid analysis) but in addition, expected results with respect to particular inputs in the resource domains would also be specified. The goal of the analyzer would then be to verify whether the properties are valid or not and also to ensure that the results are the expected ones. This, of course, would necessitate very precise information from the analyst. This testing strategy combined with, for example, "design by contract" specifications for program behaviour (invariants, pre- and post-conditions), appears to hold promise as a practical test strategy based on formal methods.

4.4 Worst-Case Hybrid Analysis

Worst-case hybrid analysis would be an effort towards asking as few inputs as possible of the analyst. Therefore, the analyst would not have to answer interactive questions or define resource domains and test cases in addition to the properties to verify. The analysis process would rather use a powerful static analysis in order to determine the worst-case scenarios, i.e., the most conservative and critical ones, and would then inform the analyst of its results and conclusions in a way that could be used to assess the potential damage of the verified program on the system with respect to the defined properties. Such a verification would certainly be of interest in situations when a preliminary investigation must be conducted on a small to medium-sized piece of software (under approximately 20K lines of Java source code).

4.5 Statically-Supported Dynamic Analysis

The hybrid analysis approaches presented in the previous subsections were all intertwined with static analyses (usually, dataflow and dependency analyses) in order to further investigate the proof of the satisfiability of the properties to be verified. The following dynamic analysis approach is also intensively supported by static analysis, that is, its dynamic analysis algorithm becomes intertwined with static analysis (once again, the static analysis used is anticipated to predominantly be dataflow analysis). This approach has been given the name *statically-supported dynamic analysis*.

It essentially consists of the following:

1. Perform a pre-execution static analysis and then, during execution:
   (a) Loop
      i. Detect and memorize information that can make the static analysis progress
      ii. Continue the static analysis with the additional information

It would be of great interest to define how the results of the static analysis could be represented in order to facilitate the dynamic analysis at runtime.

Moreover, and most importantly, such a statically-supported dynamic analysis could potentially detect where a property could (or will) be violated in order to react more rapidly than would have been possible without statically-collected information. Therefore, statically-supported dynamic analysis could hypothetically not suffer from too-late diagnosis, as can be the case for traditional dynamic analysis.

This last approach is thus another manner of tackling hybrid analysis. Approaches based on static analysis are augmented by dynamic analysis principles. On the other hand, this last approach is dynamic analysis augmented by static analysis principles.

5 Small Case Study

This section presents a small case study in order to concretely illustrate the concepts and theories explained in this document.

5.1 Java Source Code

Here is the Java source code of a small sample program:

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

```
public class Example {
  public static void main(String[] args)
    throws IOException
  {
    // Opening file "file1.txt"
    BufferedInputStream bis = new BufferedInputStream(
                            new FileInputStream("file1.txt"));

    // Closing file "file1.txt"
    bis.close();

    // Getting standard input (keyboard)
    BufferedReader in = new BufferedReader(
                      new InputStreamReader(System.in));

    // Asking for a filename to "process" (delete)
    System.out.print("Please enter a filename to process: ");

    // Saving the entered filename
    String filename = in.readLine();

    // Creating an abstract representation of the file
    File file = new File(filename);

    // "Processing" (deleting) the file
    if (file.delete())
    {
      System.out.println(filename + " successfully processed.");
    }
    else
    {
      System.out.println("Could not process " + filename + ".");
    }
  }
}
```

As the trained Java programmer eye can see, this program is very simple. Firstly, it opens a file named “file1.txt” and immediately closes it. This segment of the program is going to be used only to discuss what happens when a security-critical resource like a file is statically present in the code.

Secondly, the program asks the user to type the name of a file that is going to be "processed". In fact, the process performed on the given file is that it is going to be deleted. This segment of the program is going to be used to discuss what happens when a security-critical resource is not statically present in the code but dynamically fetched by the program. Depending on whether the program could delete the given file or not, it gives appropriate feedback to the user.

5.2 Java Bytecode Instructions

The bytecode instructions of the small example program, as generated by SUN's javap tool, are as follows:

```
public class Example extends java.lang.Object {
    public Example();
    /*   ()V   */
    /* Stack=1, Locals=1, Args_size=1 */
    public static void main(java.lang.String[]) throws java.io.IOException;
    /*   ([Ljava/lang/String;)V   */
    /* Stack=5, Locals=5, Args_size=1 */
}

Method Example()
   0 aload_0
   1 invokespecial #1 <Method java.lang.Object()>
   4 return

Method void main(java.lang.String[])
   0 new #2 <Class java.io.BufferedInputStream>
   3 dup
   4 new #3 <Class java.io.FileInputStream>
   7 dup
   8 ldc #4 <String "file1.txt">
  10 invokespecial #5 <Method java.io.FileInputStream(java.lang.String)>
  13 invokespecial #6 <Method java.io.BufferedInputStream(java.io.InputStream)>
  16 astore_1
  17 aload_1
  18 invokevirtual #7 <Method void close()>
  21 new #8 <Class java.io.BufferedReader>
  24 dup
  25 new #9 <Class java.io.InputStreamReader>
  28 dup
  29 getstatic #10 <Field java.io.InputStream in>
  32 invokespecial #11 <Method java.io.InputStreamReader(java.io.InputStream)>
  35 invokespecial #12 <Method java.io.BufferedReader(java.io.Reader)>
  38 astore_2
  39 getstatic #13 <Field java.io.PrintStream out>
  42 ldc #14 <String "Please enter a filename to process: ">
```

```
 44 invokevirtual #15 <Method void print(java.lang.String)>
 47 aload_2
 48 invokevirtual #16 <Method java.lang.String readLine()>
 51 astore_3
 52 new #17 <Class java.io.File>
 55 dup
 56 aload_3
 57 invokespecial #18 <Method java.io.File(java.lang.String)>
 60 astore 4
 62 aload 4
 64 invokevirtual #19 <Method boolean delete()>
 67 ifeq 98
 70 getstatic #13 <Field java.io.PrintStream out>
 73 new #20 <Class java.lang.StringBuffer>
 76 dup
 77 invokespecial #21 <Method java.lang.StringBuffer()>
 80 aload_3
 81 invokevirtual #22 <Method java.lang.StringBuffer append(java.lang.String)>
 84 ldc #23 <String " successfully processed.">
 86 invokevirtual #22 <Method java.lang.StringBuffer append(java.lang.String)>
 89 invokevirtual #24 <Method java.lang.String toString()>
 92 invokevirtual #25 <Method void println(java.lang.String)>
 95 goto 128
 98 getstatic #13 <Field java.io.PrintStream out>
101 new #20 <Class java.lang.StringBuffer>
104 dup
105 invokespecial #21 <Method java.lang.StringBuffer()>
108 ldc #26 <String "Could not process ">
110 invokevirtual #22 <Method java.lang.StringBuffer append(java.lang.String)>
113 aload_3
114 invokevirtual #22 <Method java.lang.StringBuffer append(java.lang.String)>
117 ldc #27 <String ".">
119 invokevirtual #22 <Method java.lang.StringBuffer append(java.lang.String)>
122 invokevirtual #24 <Method java.lang.String toString()>
125 invokevirtual #25 <Method void println(java.lang.String)>
128 return
```

The method that is of interest here is the one identified by

`Method void main(java.lang.String[]).`

The one identified by `Method Example()` is simply the class default constructor. It is automatically generated by the compiler.

With these bytecode instructions, it is possible to automatically generate the corresponding model, as defined in section 2. Being quite intuitive, this algorithm is not presented in this document for the sake of brevity. Rather, the model is directly presented in the following subsection.

5.3 Model

The model that corresponds to the small example program is formally defined as a tuple $(S, \Sigma, \rho, \iota, F)$, where

- $S = \{s_0, s_3, s_4, s_7, \ldots, s_{122}, s_{125}, s_{128}, s_{129}\}$,

- $\Sigma = \{($new, <Class java.io.BufferedInputStream>$)$,
  $($dup, $\lambda)$,
  $($new, <Class java.io.FileInputStream>$)$,
  $($ldc, <String "file1.txt">$)$,
  $($invokespecial, <Method java.io.FileInputStream(java.lang.String)>$)$,
  $($invokespecial, <Method java.io.BufferedInputStream(java.io.InputStream)>$)$,
  $($astore_1, $\lambda)$,
  $($aload_1, $\lambda)$,
  $($invokevirtual, <Method void close()>$)$,
  $($new, <Class java.io.BufferedReader>$)$,
  $($new, <Class java.io.InputStreamReader>$)$,
  $($getstatic, <Field java.io.InputStream in>$)$,
  $($invokespecial, <Method java.io.InputStreamReader(java.io.InputStream)>$)$,
  $($invokespecial, <Method java.io.BufferedReader(java.io.Reader)>$)$,
  $($astore_2, $\lambda)$,
  $($getstatic, <Field java.io.PrintStream out>$)$,
  $($ldc, <String "Please enter a filename to process: ">$)$,
  $($invokevirtual, <Method void print(java.lang.String)>$)$,
  $($aload_2, $\lambda)$,
  $($invokevirtual, <Method java.lang.String readLine()>$)$,
  $($astore_3, $\lambda)$,
  $($new, <Class java.io.File>$)$,
  $($aload_3, $\lambda)$,
  $($invokespecial, <Method java.io.File(java.lang.String)>$)$,
  $($astore 4, $\lambda)$,
  $($aload 4, $\lambda)$,
  $($invokevirtual, <Method boolean delete()>$)$,
  $($ifeq, $s_{98})$,
  $($new, <Class java.lang.StringBuffer>$)$,
  $($invokespecial, <Method java.lang.StringBuffer()>$)$,
  $($invokevirtual, <Method java.lang.StringBuffer append(java.lang.String)>$)$,
  $($ldc, <String " successfully processed.">$)$,
  $($invokevirtual, <Method java.lang.String toString()>$)$,
  $($invokevirtual, <Method void println(java.lang.String)>$)$,
  $($goto, $s_{128})$,
  $($ldc, <String "Could not process ">$)$,
  $($ldc, <String ".">$)$,
  $($return, $\lambda)\}$,

- $\rho = \{(s_0, ($new, <Class java.io.BufferedInputStream>$), s_3)$,
  $(s_3, ($dup, $\lambda), s_4)$,
  $(s_4, ($new, <Class java.io.FileInputStream>$), s_7)$,
  $(s_7, ($dup, $\lambda), s_8)$,
  $(s_8, ($ldc, <String "file1.txt">$), s_{10})$,
  $(s_{10}, ($invokespecial, <Method java.io.FileInputStream(java.lang.String)>$), s_{13})$,
  $(s_{13}, ($invokespecial, <Method java.io.BufferedInputStream(java.io.InputStream)>$), s_{16})$,
  $(s_{16}, ($astore_1, $\lambda), s_{17})$,
  $(s_{17}, ($aload_1, $\lambda), s_{18})$,
  $(s_{18}, ($invokevirtual, <Method void close()>$), s_{21})$,
  $(s_{21}, ($new, <Class java.io.BufferedReader>$), s_{24})$,
  $(s_{24}, ($dup, $\lambda), s_{25})$,
  $(s_{25}, ($new, <Class java.io.InputStreamReader>$), s_{28})$,
  $(s_{28}, ($dup, $\lambda), s_{29})$,
  $(s_{29}, ($getstatic, <Field java.io.InputStream in>$), s_{32})$,
  $(s_{32}, ($invokespecial, <Method java.io.InputStreamReader(java.io.InputStream)>$), s_{35})$,

(s_{35}, (invokespecial, <Method java.io.BufferedReader(java.io.Reader)>), s_{38}),
(s_{38}, (astore_2, λ), s_{39}),
(s_{39}, (getstatic, <Field java.io.PrintStream out>), s_{42}),
(s_{42}, (ldc, <String "Please enter a filename to process: ">), s_{44}),
(s_{44}, (invokevirtual, <Method void print(java.lang.String)>), s_{47}),
(s_{47}, (aload_2, λ), s_{48}),
(s_{48}, (invokevirtual, <Method java.lang.String readLine()>), s_{51}),
(s_{51}, (astore_3, λ), s_{52}),
(s_{52}, (new, <Class java.io.File>), s_{55}),
(s_{55}, (dup, λ), s_{56}),
(s_{56}, (aload_3, λ), s_{57}),
(s_{57}, (invokespecial, <Method java.io.File(java.lang.String)>), s_{60}),
(s_{60}, (astore 4, λ), s_{62}),
(s_{62}, (aload 4, λ), s_{64}),
(s_{64}, (invokevirtual, <Method boolean delete()>), s_{67}),
(s_{67}, (ifeq, s_{98}), s_{70}),
(s_{70}, (getstatic, <Field java.io.PrintStream out>), s_{73}),
(s_{73}, (new, <Class java.lang.StringBuffer>), s_{76}),
(s_{76}, (dup, λ), s_{77}),
(s_{77}, (invokespecial, <Method java.lang.StringBuffer()>), s_{80}),
(s_{80}, (aload_3, λ), s_{81}),
(s_{81}, (invokevirtual, <Method java.lang.StringBuffer append(java.lang.String)>), s_{84}),
(s_{84}, (ldc, <String " successfully processed.">), s_{86}),
(s_{86}, (invokevirtual, <Method java.lang.StringBuffer append(java.lang.String)>), s_{89}),
(s_{89}, (invokevirtual, <Method java.lang.String toString()>), s_{92}),
(s_{92}, (invokevirtual, <Method void println(java.lang.String)>), s_{95}),
(s_{67}, (ifeq, s_{98}), s_{98}),
(s_{98}, (getstatic, <Field java.io.PrintStream out>), s_{101}),
(s_{101}, (new, <Class java.lang.StringBuffer>), s_{104}),
(s_{104}, (dup, λ), s_{105}),
(s_{105}, (invokespecial, <Method java.lang.StringBuffer()>), s_{108}),
(s_{108}, (ldc, <String "Could not process ">), s_{110}),
(s_{110}, (invokevirtual, <Method java.lang.StringBuffer append(java.lang.String)>), s_{113}),
(s_{113}, (aload_3, λ), s_{114}),
(s_{114}, (invokevirtual, <Method java.lang.StringBuffer append(java.lang.String)>), s_{117}),
(s_{117}, (ldc, <String ".">), s_{119}),
(s_{119}, (invokevirtual, <Method java.lang.StringBuffer append(java.lang.String)>), s_{122}),
(s_{122}, (invokevirtual, <Method java.lang.String toString()>), s_{125}),
(s_{125}, (invokevirtual, <Method void println(java.lang.String)>), s_{128}),
(s_{95}, (goto, s_{128}), s_{128}),
(s_{128}, (return, λ), s_{129})},

- $\iota = s_0$, and
- $F = \{s_{129}\}$.

5.3.1 Remarks

A few remarks concerning this model:

1. Because program points lie between each bytecode instruction of the program, the states in S are defined from the bytecode instruction offsets, hence $S = \{s_0, s_3, s_4, s_7, \ldots, s_{122}, s_{125}, s_{128}, s_{129}\}$. The state s_{129} is necessary because it models the program exit point.
2. For the sake of simplicity of future processing performed on the model, symbolic pointers to the constant pool are resolved in the model. Therefore, bytecode instructions like `new #2` are translated to

(`new`, `<Class java.io.BufferedInputStream>`).

Tools like SUN's javap already perform constant pool symbolic pointers resolving.

3. For the same reason, symbolic pointers to bytecode offsets are also resolved in terms of the model states. For example, bytecode instructions like `ifeq 98` and `goto 128` are translated to (`ifeq`, s_{98}) and (`goto`, s_{128}), in the current model.

4. This model contains two interesting functions (in the sense defined in the logic, see section 3), i.e., bytecode instructions that call security-critical methods. They are:

   (a) (`invokespecial`, `<Method java.io.FileInputStream(java.lang.String)>`) and

   (b) (`invokevirtual`, `<Method boolean delete()>`).

   The first one corresponds to the opening of a file and the second one, to the deletion of a file. It is easy to automatically find these functions of interest by inspecting the model and comparing it to the list of security-critical methods listed in AP, which is calculated by the technique explained in subsection 3.1.1. To fit with the theory of section 3, these functions should conceptually be viewed as openfile("file1.txt") and deletefile, respectively. In fact, this model should pass through an abstraction process that performs such label translations. For the sake of brevity, this process is not presented in this document.

5.4 Logic

Once the model of the small example program has been generated and abstracted, it is possible to define properties that can be verified. These properties are defined by using the logic detailed in section 3. For this small case study, two properties are defined and explained in the following subsections.

5.4.1 Property #1: Do Not Open File "file1.txt"

Suppose that it is necessary to ensure that the example program does not open the file named "file1.txt". Of course, it could be any file and not necessarily "file1.txt". This property is expressed as the following formula in the logic:

$$\neg\text{openfile}(\text{“file1.txt”}),$$

where openfile is an element of the set AP and "file1.txt", of R.

It is now time to verify if the program respects this property. It boils down to determining if $M \models \Phi$, where M is the model of subsection 5.3 (considering it abstracted as of remark 4 of subsection 5.3.1) and $\Phi = \neg$openfile(“file1.txt”). The algorithm used to perform this calculation corresponds to the semantics of the logic, described in subsection 3.2. The execution trace for the current property is detailed below:

1. Since Φ is a negation, the following rule is used first:

$$M \models \neg\Phi \text{ iff } \text{not}(M \models \Phi)$$

2. $M \models \Phi$ must therefore be calculated. Since this time

$$\Phi = \text{openfile(“file1.txt”)}$$

   and openfile is an atomic proposition p, the following rule is used:

```
$M \models p(s)$   iff   if $s = \lambda$ then
                            there exists $s_1, s_2 \in S$ and $(q, r) \in \Sigma$
                                such that $(s_1, (q, r), s_2) \in \rho$ and $p = q$
                         else if $s \neq \lambda$ then
                            there exists $s_1, s_2 \in S$ such that $(s_1, (p, s), s_2) \in \rho$ or
                            if there exists $s_3, s_4 \in S$
                                    such that $(s_3, (p, \lambda), s_4) \in \rho$ then
                                Hybrid_Analysis()
```

3. Here, $s \neq \lambda$ and there exists $s_1, s_2 \in S$ such that $(s_1, (p, s), s_2) \in \rho$. Indeed, it suffices to set $s_1 = s_{10}$ and $s_2 = s_{13}$. Therefore, it is concluded that $M \models \Phi$ is *true*, where $\Phi =$ openfile(“file1.txt”).

4. Finally, going back to the first rule used for $M \models \Phi$, where

$$\Phi = \neg\text{openfile(“file1.txt”)},$$

   it is concluded that it is *false* because not(*true*) = *false*.

The verification of this property does not involve hybrid analysis because the atomic proposition present in the property could be directly found in the model. The following example, however, involves hybrid analysis.

5.4.2 Property #2: Do Not Delete File "critical.doc"

Suppose again that it is necessary to ensure that the example program does not delete the file named "critical.doc". Of course, it could be any file and not necessarily "critical.doc". This property is expressed as the following formula in the logic:

$$\neg\text{deletefile}(\text{“critical.doc”}),$$

where deletefile is an element of the set AP and "critical.doc", of R.

It is now time to verify if the program respects this property. It boils down to determining if $M \models \Phi$, where M is the model of subsection 5.3 (considering it abstracted as of remark 4 of subsection 5.3.1) and $\Phi = \neg\text{deletefile}(\text{“critical.doc”})$. Once again, the algorithm used to perform this calculation corresponds to the semantics of the logic, described in subsection 3.2. The execution trace for the current property is detailed below:

1. The trace starts exactly like in the previous example. Since Φ is a negation, the following rule is used first:

   $$M \models \neg\Phi \text{ iff not}(M \models \Phi)$$

2. $M \models \Phi$ must therefore be calculated. Since this time

   $$\Phi = \text{deletefile}(\text{“critical.doc”})$$

   and deletefile is an atomic proposition p, the following rule is used:

   ```
   $M \models p(s)$   iff   if $s = \lambda$ then
                               there exists $s_1, s_2 \in S$ and $(q, r) \in \Sigma$
                                   such that $(s_1, (q, r), s_2) \in \rho$ and $p = q$
                            else if $s \neq \lambda$ then
                               there exists $s_1, s_2 \in S$ such that $(s_1, (p, s), s_2) \in \rho$ or
                               if there exists $s_3, s_4 \in S$
                                       such that $(s_3, (p, \lambda), s_4) \in \rho$ then
                                   Hybrid_Analysis()
   ```

3. Here, $s \neq \lambda$ but there does not exist $s_1, s_2 \in S$ such that $(s_1, (p, s), s_2) \in \rho$. However, there exists $s_3, s_4 \in S$ such that $(s_3, (p, \lambda), s_4) \in \rho$. Indeed,

it suffices to set $s_3 = s_{64}$ and $s_4 = s_{67}$. In conventional static analysis, the chances are that this would be sufficient to conclude that M *does* satisfy Φ, where $\Phi =$ deletefile(“critical.doc”). Therefore, M would *not* satisfy Φ, when $\Phi = \neg$deletefile(“critical.doc”). However, as noted before, this result could be overly imprecise. Indeed, by using one of the approaches suggested in section 4, it could be possible to conclude that, in the current environment, this property is actually *true*. Therefore, in the current framework, it is concluded that hybrid analysis must be used, and thus the help of the analyst, by using interaction or configuration files, for instance, is needed in order to try to provide more precise analysis results.

4. The approach to hybrid analysis that is currently being studied is described on this example in the following subsection.

5.4.3 Generalization of These Properties

The security properties presented in the preceding subsections are purposely very simple. They are used for the sole purpose of explaining the basic concepts of the logic. However, this logic offers a rich syntax for constraining applications. Once the normal extents of network access, process creation/deletion, and so on are precisely defined and added to the logic, complex properties could be defined. The potential for malicious or inadvertent erroneous program execution will then be greatly reduced.

5.5 Hybrid Analysis

As seen in the example of subsection 5.4.2, hybrid analysis is needed where conventional static analysis would normally yield over-approximated results. This subsection gives additional details on the approach to hybrid analysis that is currently being studied by using an example. This approach is a subset of interactive hybrid analysis, as described in subsection 4.1. What is currently set aside for future study is the analysis that should be performed to determine which abstract actions should be refined as a priority, depending on the benefits of refining them. In other words, questions are not asked optimally, in terms of the anticipated benefits for the analysis and the minimal annoyance for the analyst. This makes the current approach less practical but it is used to set the stage for future improvements.

The current hybrid analysis approach follows five main steps, explained in the following subsections:

1. Perform a data dependency analysis on the method that is being analyzed and for which information is missing,

2. determine the data source (i.e., keyboard, file, network, etc.) of the missing information,

3. present the situation to the analyst,

4. ask an appropriate question to the analyst in order to obtain the missing information, and

5. use the answer to continue the analysis.

5.5.1 Performing Data Dependency Analysis

In order to be able to present the situation to the analyst and to ask an appropriate question to ascertain missing information, a data dependency analysis must be performed on the method being analyzed. Such analysis is able to trace the interdependencies between pieces of data in a program and determine from which source they originate. In the current framework, the data dependency analysis is used to determine how the missing information is actually obtained by the program at runtime. This will be of great help when it is time to ask the analyst to play the role of the program and furnish this missing information.

Data dependency analyses are tailored to specific execution architectures and instruction sets. The current analysis targets Java bytecode instructions. This means that it must take into account the operand stack and registers of the Java virtual machine and how it makes use of them. Data dependency analysis needs to use the results of a dataflow analysis performed on the Java bytecode instructions. This dataflow analysis takes into account the operand stack and registers for the data dependency analysis. Thus, it has to be executed prior to the data dependency analysis itself.

At the time of writing this document, neither the dataflow nor the data dependency analyses have been completely defined and formalized for Java bytecode instructions. Therefore, complete algorithms for these analyses are not presented. However, the key ideas behind these algorithms are presented in the following subsections.

5.5.1.1 Dataflow Analysis

The dataflow analysis consists of simulating the flow of data manipulated by the program. It is first performed on each method individually (this technique is called intra-procedural dataflow analysis) and then on the entire program (called inter-procedural dataflow analysis), considering the results obtained during the intra-procedural dataflow analysis.

The algorithm of the intra-procedural dataflow analysis follows this general behavior (technical details are intentionally omitted in this document; please refer to [3] for a more detailed description):

```
1. Initialization
   (a) For each opcode $o$ of the current method
       i. Set the modified flag of $o$ to false
       ii. Set the dataflow information associated with $o$ to unknown
   (b) Set the modified flag of the first opcode of the current method to true
   (c) Set the dataflow information associated with the first opcode of the current
       method to the information provided by the current method signature
2. Main loop
   (a) While there is an opcode $o$ with a modified flag set to true in the current
       method
       i. Set the modified flag of $o$ to false
       ii. Simulate the execution of $o$ on its associated dataflow information. This
           consists of consuming the operands of $o$ and producing its output.
       iii. For each successor opcode $s$ of $o$
            A. Merge the dataflow information associated with $o$ into the informa-
               tion associated with $s$. Call the resulting dataflow information $r$.
            B. If $r$ is not equal to the dataflow information associated with $s$
               • Set the dataflow information associated with $s$ to $r$
               • Set the modified flag of $s$ to true
```

At this stage, the general behavior of the inter-procedural dataflow analysis has not been fully elaborated and will not be presented in this document.

5.5.1.2 Data Dependency Analysis

Once the dataflow analysis has been performed, the operands (at least their types) of each opcode are known. The data dependency analysis is then able to determine the dependence graph of these operands. In fact, this dependence graph does not have to be calculated for all operands but only for the ones of interest, that is, the operands of the bytecode instructions that call security-critical methods involved in the property to verify. Note that the difference between a bytecode instruction *parameter* and a bytecode instruction *operand* is that the parameter is statically determined and is present in the bytecode itself. However, the operand is dynamically determined and is taken at run-time from the operand stack or registers.

In the example of subsection 5.4.2, the only bytecode instruction is

```
64 invokevirtual #19 <Method boolean delete()>.
```

It takes only one operand: the object on which the method is being called. The data dependency analysis would thus determine where the operand of this instruction comes from, i.e., its dependence graph up to the point where it is defined by the program. For the example being studied, the data dependency analysis determines that the operand is taken from register 4 and that the value in register 4 is in fact the output of the opcode

```
57 invokespecial #18 <Method java.io.File(java.lang.String)>.
```

The analysis also determines that this opcode takes its two operands from the stack and register 3, and execution continues, until it reaches opcode

```
29 getstatic #10 <Field java.io.InputStream in>,
```

where it can conclude that, in fact, the operand of the bytecode instruction of interest (at offset 64) is a file whose name is defined by the user of the program entering text on the keyboard (the standard input).

In this manner, the data dependency analysis determines the source of the data on which security-critical methods are called in the program being verified.

5.5.2 Presenting the Situation

The sources of the data on which security-critical methods are called in the program being verified can now be presented to the analyst in a more readily comprehensible manner. In fact, at this point, all the gathered information can be presented. That is:

1. The property being verified is *not* necessarily *true* and *not* necessarily *false*.
2. The information needed to determine the truth value of the property with certainty is missing in the program *but* it is defined at run-time by known sources.

For the example of subsection 5.4.2, this information would be summarized by a graph (in fact, a sequence in this particular case) illustrating the fact that, at run-time, the keyboard (or standard input within a command pipe-line) actually defines the name of the file being deleted.

5.5.3 Asking the Appropriate Question

Once the analyst has been informed of the situation, it is time to ask questions. This is the *hybrid* component of the analysis, that is, the part inspired from dynamic analysis principles. The particular approach used at this time is mostly inspired from conventional debugging: interact with the person who is most probably able to furnish the missing information.

In the current framework, with such an inexpressive logic (it does not even consider the program execution paths or sequences like temporal and modal logics do), there are not many questions that can be asked in order to get help from the analyst. In fact, probably the most reasonable question to ask is:

> "Considering the situation and its inherent risks, do you think that the property will be satisfied or not?"

However, and most importantly, considering the data sources of the security-critical method calls implied in the property to verify, the analyst can now judge appropriately whether to execute the program in an environment where these sources are under control, in a monitored environment, for instance.

Of course, once a property has been determined to be acceptable, another one can be immediately verified and the hybrid analysis goes on.

6 Discussion

Hybrid analysis, as described in this document, is a new analysis that combines static and dynamic analysis principles in order to provide more precise results than conventional static analysis [4]. One could say that this added precision comes at the price of unsoundness. This is not necessarily true. In fact, in general, it is believed that hybrid analysis requires a *trade-off* between precision and soundness [4]; to some extent, it can be both precise *and* sound.

It is actually the case for the hybrid analysis described in this document. Indeed, the "sound part" of this analysis is provided by the semantics of the logic and the dataflow and dependency algorithms. The "precise part" is provided by the interaction with the analyst. Moreover, in other approaches based on parameterized, test-based, or worst-case hybrid analyses (refer to section 4), the precision could be even better, with the same sound foundation.

As described in this document, *the current framework must be extended in order to be more practical*. First, the model can easily represent non-determinism but it cannot model (true) parallelism. Even pseudo-parallelism (interleaving) between a few processes would make its number of states explode. Therefore, the model will have to evolve towards using more expressive computational theories in order to be able to represent realistic programs, such as critical military information systems. One possible theory under consideration is Abstract State Machines (ASMs). For the sake of brevity, this theory is not covered in this document (refer to [5] and [6]).

Second, the logic is not expressive enough to define really interesting properties to verify. As defined in this document, its semantics essentially corresponds to searching through a graph for particular keywords, which are security-critical method calls in this case. In practice, it should be possible to go further and specify properties like this one:

> "No confidential files are copied.".

Such properties depend on many security-critical method calls and on their interaction. They cannot be specified with the current logic but they can be specified with temporal (like CTL) or modal logics (like the μ-calculus). Again, the current framework will evolve towards using such logics in order to be more appropriate for critical military information systems.

Finally, the approach to hybrid analysis currently investigated in this document also needs to be extended. Systematically questioning the analyst each time a too abstract security-critical method call is encountered in the model is simply unrealistic in practice when dealing with large programs and complex properties. Therefore, while investigating more automated approaches is really attractive (refer to section 4), answering the four questions of subsection 4.1 and performing an analysis to prioritize the actions that need to be refined are musts for the practicality of the approach. This is going to be done as future work.

Hybrid analysis appears to hold great promise for improving the quality and the speed of validation and verification of critical systems. In particular, a thorough static analysis combined with testing scripts for run-time inputs appears to be a practically realizable verification strategy. High quality toolkits for generating test inputs already exist, and combining these with intra- and inter-module static analyses should enable very high assurance level validation and verification for unit testing and integration testing respectively.

7 Conclusion

Static and dynamic analysis can be combined to create a new analysis technique called hybrid analysis. This new technique can be seen as both static analysis complemented by dynamic analysis for improved precision (like in this document) or dynamic analysis complemented by static analysis for improved generality over many program executions. It is currently being developed to ensure the security of Java programs in the context of critical military information systems. By integrating advanced mechanisms to enforce security policies into one consolidated analysis and by allowing these mechanisms to communicate and cooperate, hybrid analysis greatly reduces the chances of executing malicious code.

The approach described in this document is particularly appealing because it defines a trade-off between soundness and precision that seems to be intuitively correct: *use sound static analysis to determine when it is time to use precise dynamic analysis*.

Security policy enforcement for critical military information systems is a very difficult problem. Subtle interactions between individually secure modules can result in security anomalies. Intra- and inter-module static analyses combined with testing scripts can potentially uncover these flaws and facilitate enforcement of information system security policies.

Future research work in this project promises many interesting challenges. This should stimulate discussion and encourage academic or governmental collaboration.

References

1. Brookshear, J. Glenn (1989). Theory of Computation: Formal Languages, Automata, and Complexity, Benjamin-Cummings Publishing Company. 1

2. Emerson, E. A. (1990). Temporal and modal logic. In *Handbook of Theoretical Computer Science*, Vol. B of *Formal Models and Semantics*, pp. 995–1072. Elsevier Science Publishers B.V. 3

3. Debbabi, Mourad, Desharnais, Jules, Fourati, Myriam, Menif, Emna, Painchaud, Frédéric, and Tawbi, Nadia (2002). Secure Self-Certified Code for Java. In *Formal Aspects of Security (FASec)*, London, UK. 24

4. Ernst, Michael D. (2003). Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pp. 24–27. 26

5. Gurevich, Yuri (1995). Evolving Algebras 1993: Lipari Guide. In Börger, Egon, (Ed.), *Specification and Validation Methods*, pp. 231–243. Oxford University Press. 27

6. Börger, Egon and Stärk, Robert (2003). Abstract State Machines: A Method for High-Level System Design and Analysis, Springer-Verlag. http://www.di.unipi.it/AsmBook/. 27

List of Acronyms

API	Application Programming Interface
ASM	Abstract State Machine
C2IS	Command and Control Information Systems
DRDC	Defence Research and Development Canada

Glossary

Dynamic analysis	A set of techniques to analyze software during execution.
Formal methods	A set of hardware and software analysis methods that are based on mathematical formalism, symbolism, and logic.
Java Security Manager	A component of the Java Virtual Machine that is responsible to ensure high level security.
Java Virtual Machine	The main component of the Java Platform that is responsible for safely and securely executing Java programs.
Model checking	A technique used to verify that a property holds on every possible state of a hardware or software system.
Static analysis	A set of techniques to analyze software prior to execution.
Validation	A process that is used to ensure that the design requirements are met in a product being developed.
Verification	A process that is used to ensure that a product being developed meets some predefined quality standards.

Distribution list

Internal Distribution
DRDC Valcartier TM 2004-060

1	-	Director General
3	-	Document Library
1	-	F. Painchaud (author)

External Distribution
DRDC Valcartier TM 2004-060

1	-	Directorate R & D – Knowledge and Information Management (PDF file)

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when document is classified)

1.	ORIGINATOR (the name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Centre sponsoring a contractor's report, or tasking agency, are entered in section 8.) Defence R & D Canada – Valcartier 2459 Pie-XI Blvd North, Québec, QC, Canada, G3J 1X5	2.	SECURITY CLASSIFICATION (overall security classification of the document including special warning terms if applicable). UNCLASSIFIED		
3.	TITLE (the complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S,C,R or U) in parentheses after the title). Java hybrid analysis version 0.5				
4.	AUTHORS (Last name, first name, middle initial. If military, show rank, e.g. Doe, Maj. John E.) Painchaud, F.				
5.	DATE OF PUBLICATION (month and year of publication of document) February 2007	6a.	NO. OF PAGES (total containing information. Include Annexes, Appendices, etc). 40	6b.	NO. OF REFS (total cited in document) 6
7.	DESCRIPTIVE NOTES (the category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered). Technical Memorandum				
8.	SPONSORING ACTIVITY (the name of the department project office or laboratory sponsoring the research and development. Include address). Defence R & D Canada – Valcartier 2459 Pie-XI Blvd North, Québec, QC, Canada, G3J 1X5				
9a.	PROJECT OR GRANT NO. (if appropriate, the applicable research and development project or grant number under which the document was written. Specify whether project or grant). 15BF34	9b.	CONTRACT NO. (if appropriate, the applicable number under which the document was written).		
10a.	ORIGINATOR'S DOCUMENT NUMBER (the official document number by which the document is identified by the originating activity. This number must be unique.) DRDC Valcartier TM 2004-060	10b.	OTHER DOCUMENT NOs. (Any other numbers which may be assigned this document either by the originator or by the sponsor.)		
11.	DOCUMENT AVAILABILITY (any limitations on further dissemination of the document, other than those imposed by security classification) (X) Unlimited distribution ( ) Defence departments and defence contractors; further distribution only as approved ( ) Defence departments and Canadian defence contractors; further distribution only as approved ( ) Government departments and agencies; further distribution only as approved ( ) Defence departments; further distribution only as approved ( ) Other (please specify):				
12.	DOCUMENT ANNOUNCEMENT (any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution beyond the audience specified in (11) is possible, a wider announcement audience may be selected).				

13. ABSTRACT (a brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual).

This document proposes a new program analysis technique, called *hybrid analysis*, that combines both static and dynamic analysis principles. It is currently being developed to ensure the security of Java programs in the context of critical military information systems. This new program verification technique is theoretically sound and precise, which constitutes a major contribution to the formal program verification domain.

At this stage, Java hybrid analysis is based on finite state automata theory and first-order predicate logic, only succinctly covered in this document. The reader is thus assumed to be relatively familiar with these domains. However, the precise model and logic used by Java hybrid analysis are extensively described. Furthermore, five different hybrid analysis approaches have been devised. They have been given the following names: *interactive hybrid analysis*, *parameterized hybrid analysis*, *test-based hybrid analysis*, *worst-case hybrid analysis*, and *statically-supported dynamic analysis*. A small case study is presented in order to illustrate the concepts behind the first approach. Finally, a discussion depicts the future of this approach.

The Java hybrid analysis detailed in this document is labeled version 0.5 to depict the fact that it is still unstable and evolving.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus. e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus-identified. If it not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title).

C2IS Certification
Java
Java Security
Static Analysis
Dynamic Analysis
Hybrid Analysis