7th Workshop on Bytecode Semantics, Verification, Analysis and Transformation

Bytecode 2012

Tallinn, Estonia, 31 March 2012

Proceedings

7th Workshop on Bytecode Semantics, Verification, Analysis and Transformation

Bytecode 2012

Tallinn, Estonia, 31 March 2012

Proceedings

Institute of Cybernetics at Tallinn University of Technology

Tallinn ○ 2012

7th Workshop on Bytecode Semantics, Verification, Analysis and Transformation
Bytecode 2012
Tallinn, Estonia, 31 March 2012
Proceedings

Edited by Marieke Huisman

Institute of Cybernetics at Tallinn University of Technology
Akadeemia tee 21, 12618 Tallinn, Estonia
http://www.ioc.ee/

These proceedings of Bytecode 2012 are final.

Preface

This volume contains the proceedings of the Bytecode 2012 workshop, the seventh Workshop on Bytecode Semantics, Verification, Analysis and Transformation, held in Tallinn, Estonia, on the 31$^{\text{th}}$ of March 2012 as part of ETAPS 2012.

Bytecode, such as produced by e.g., Java and .NET compilers, has become an important topic of interest, both for industry and academia. The industrial interest stems from the fact that bytecode is typically used for Internet and mobile device applications (smart cards, phones, etc.), where security is a major issue. Moreover, bytecode is device independent and allows dynamic loading of classes, which provides an extra challenge for the application of formal methods. Also the unstructuredness of the code and the pervasive presence of the operand stack provide further challenges for the analysis of bytecode. This workshop focuses on theoretical and practical aspects of semantics, verification, analysis, certification and transformation of bytecode.

There were 6 submissions for the workshop. Each submission was reviewed by at least 3 programme committee members. The committee decided to accept 5 papers. The programme also includes 3 invited talks by Jeff Foster, Diego Garbervetsky, and James Hunt.

As the workshop chair, I would like to thank the program committee, whose invaluable help and enthusiasm ensured the success of the event. We would also like to thank all anonymous referees, for their hard work. Finally, Stefan Blom helped out preparing the informal proceedings for the workshop.

February 2012

Marieke Huisman
University of Twente
Netherlands

Workshop Organisation

Programme Chair

Marieke Huisman, University of Twente, Netherlands.

Programme Committee

Elvira Albert, Complutense University of Madrid, Spain
June Andronick, NICTA, Australia
Massimo Bartoletti, University of Cagliari, Italy
Lennart Beringer, Princeton University, USA
Francesco Logozzo, Microsoft Research, USA
Peter Müller, ETH Zürich, Switzerland
Tamara Rezk, INRIA Sophia Antipolis-Mediterranee, France
Bernhard Scholz, University of Sydney, Australia
Fausto Spoto, University of Verona, Italy

Subreviewers

Paul Subotic
Pietro Ferrara
Vasvi Kakkad

Contents

Invited Talks

Using Bytecode Transformation to Retrofit Fine-Grained Security Policies on Unmodified Android

Jeff Foster

Quantitative Analysis of Java/.Net-like Programs to Understand Heap Memory Requirements

Diego Garbervetsky

Bytecode and Safety-Critical Systems: Friend or Foe?

James Hunt

Contributed Talks

Conditional Termination of Loops over Arrays

Elvira Albert, Samir Genaim and Guillermo Román-Díez

BCT: A translator from MSIL to Boogie

Michael Barnett and Shaz Qadeer

Log-based Lazy Monitoring of OSGi Bundles

Gabriele Costa, Giulio Caravagna, Giovanni Pardini and Luca Wiegand

Extended Abstract: Embeddable Security-by-Contract Verifier for Java Card

Olga Gadyatskaya, Eduardo Lostal and Fabio Massacci

Study, Formalisation, and Analysis of Dalvik Bytecode

Henrik Søndberg Karlsen, Erik Ramsgaard Wognsen, Mads Chr. Olesen and René Rydhof Hansen

Using Bytecode Transformation to Retrofit Fine-Grained Security Policies on Unmodified Android

Jeff Foster

University of Maryland, USA

Abstract. Google's Android platform includes a permission model that protects access to sensitive capabilities, such as Internet access, GPS use, and telephony. We have found that Android's current permissions are often overly broad, providing apps with more access than they truly require. This deviation from least privilege increases the threat from vulnerabilities and malware. To address this issue, we present a novel system that can replace existing platform permissions with finer-grained ones. A key property of our approach is that it runs today, on stock Android devices, requiring no platform modifications. Moreover, we can retrofit our approach onto existing apps by transforming app bytecode to access sensitive resources through a restricted interface. We evaluated our approach on several popular, free Android apps. We found that we can replace many commonly used "dangerous" permissions with finer-grained permissions. Moreover, apps transformed to use these finer-grained permissions run largely as expected, with reasonable performance overhead.

Quantitative Analysis of Java/.Net-like Programs to Understand Heap Memory Requirements

Diego Garbervetsky

University of Buenos Aires, Argentina

Abstract. There is an increasing interest in understanding and analyzing the use of resources in software and hardware systems. Certifying memory consumption is vital to ensure safety in embedded systems as well as proper administration of their power consumption; understanding the number of messages sent through a network is useful to detect performance bottlenecks or reduce communication costs, etc. Assessing resource usage is indeed a cornerstone in a wide variety of software-intensive system ranging from embedded to Cloud computing. It is well known that inferring, and even checking, quantitative bounds is difficult (actually undecidable). Memory consumption is a particularly challenging case of resource-usage analysis due to its non-accumulative nature. Inferring memory consumption requires not only computing bounds for allocations but also taking into account the memory recovered by a GC. In this talk I will present some of the work our group have been performing in order to automatically analyze heap memory requirements. In particular, I will show some basic ideas which are core to our techniques and how they were applied to different problems, ranging from inferring sizes of memory regions in real-time Java to analyzing heap memory requirements in Java/.Net. Then, I will introduce our new compositional approach which is used to analyze (infer/verify) Java and .Net programs. Finally, I will explain some limitations of our approach and discuss some key challenges and directions for future research.

Bytecode and Safety-Critical Systems: Friend or Foe?

James Hunt

aicas GmbH, Germany

Abstract. New standards in avionics, codify the certification of systems using object-oriented technology, interpretation, garbage collection, and formal methods, provide an opportunity for using bytecode-based languages for safety-critical development. The question remains, to what extent can bytecode be used to support rather than inhibit the use of these langauges for safety-critical development. Though experience seems to indicate that using bytecode-based languages can ease the development of complex systems, the dynamic nature of these languages complicates some conventional analysis. Certainly, efforts such as BML can facilitate the application of formal analysis techniques, but more could be done. This talk will discuss some of the problems involved and present some ideas for furthering the utility of bytecode for safety-critical systems.

Conditional Termination of Loops over Arrays

Elvira Albert[1], Samir Genaim[1] and Guillermo Román-Díez[2]

[1] DSIC, Complutense University of Madrid (UCM), Spain
[2] DLSIIS, Technical University of Madrid (UPM), Spain

1 Introduction and Motivation

This paper presents a new method for proving (conditional) termination of bytecode programs that contain loops over arrays. In this section, we intuitively explain the analysis and transformations underlying our approach. For the sake of clarity, the examples in this section are written in Java, but our analysis will then be developed at the bytecode level. Fig. 1 shows two common patterns which pose new challenges to termination analysis of loops over arrays. Currently, neither Costa [4], Julia [8] or Aprove [7] can prove their (conditional) termination (when the input is not given). Note that our problem is as difficult as handling *class fields* [3]. This is because array elements can be accessed using multiple references which are aliases and besides references to arrays can change.

Let us first focus on Pattern ①. Proving its termination requires tracking the value of `a[i]`. As proposed in [2], we can try to transform `a[i]` into a *ghost* variable and then prove termination by relying on an *array-insensitive* analysis (i.e., an analysis which does not give any special treatment to arrays). The following two soundness conditions are required in order to soundly convert `a[i]` into a local variable: (1) the array reference must point to the same location during the whole execution of the loop (i.e., we cannot modify the value of `a` within the loop) and (2) all accesses to the memory location pointed by `a[i]` must be through the same reference, in this case `a[i]`. Observe that, we cannot prove (without any assumption on the input) the latter condition as `a` and `b` may refer to the same array and `i` might be equal to `j`.

Pattern ② shows a simple example of circular array traversal. Due to their better performance, circular arrays are frequently used for implementing data structures (e.g., queues or buffers). Proving termination of loops that traverse circularly arrays is challenging. First, depending on the last operation made over the array, the index of the loop can be at any arbitrary position and non-linear operations (e.g., modulo and if statements) are used to keep the index within the array dimensions. Second, due to its circular structure, termination depends on conditions on the contents of the array (i.e., array elements are typically used in the guards). In order to automatically prove termination, the following conditions must be guaranteed: (1) the searched element must have the same value at each loop iteration. In the example, the value is stored in variable `x` but, in general, it could be a constant (e.g., `null`, an integer value, etc.); (2) the array must have an element that is equal to x; and (3) similar conditions to those of pattern ① must hold in order to be able to track the value of `a[i]`.

<table>
<tr><th>Pattern ①</th><th>Program ③</th><th>Program ④</th></tr>
<tr><td><pre>while(a[i] > 0) {
  a[i]--;
  b[j]++;
}</pre></td><td><pre>while(g_1 > 0) {
  g_1--;
  g_1++;
} // (a=b ∧ i=j)</pre></td><td><pre>while(g_1 > 0){
  g_1--;
  g_2++;
} // (a≠b) ∨ (i≠j)</pre></td></tr>
<tr><th>Pattern ②</th><th colspan="2">Program ⑤</th></tr>
<tr><td><pre>while(x ≠ a[i]) {
  i = (i + 1) % a.length;
}</pre></td><td colspan="2"><pre>while(x ≠ a[i] && g ≠ i &&
      $0 \le g < a.length$) {
  i = (i + 1) % a.length;
} // $\exists g \in [0..a.length-1].\ a[g] = x$</pre></td></tr>
</table>

Fig. 1. Common patterns (conditions for termination in comments)

This paper proposes the following approach to proving termination of loops over arrays. (1) First, we develop a static analysis which allows us to obtain the *access paths* to the arrays and array elements for each of the loops, as well as constancy information on the variables of interest. As arrays are often accessed using indexes that involve arithmetic expressions (e.g., `a[i+1]`), our abstract domain contains elements for representing such expressions. Our analysis generalizes the reference constancy analysis of [2], which infers information only on class fields, to consider arrays, integer variables and arithmetic expressions. (2) By relying on the analysis information, we can automatically check if the conditions for soundly transforming array accesses into ghost variables accesses hold. If they unconditionally hold, the transformation is carried out and termination is proven by an array-insensitive termination analysis. (3) It often happens that the conditions do not hold. In such cases, we try to infer under which conditions termination can be proven. The main idea here is to consider all possible situations and prove their termination separately.

The second column of Fig. 1 shows the transformed programs for which termination analysis can be carried out by *array-insensitive* termination analyzer. For Pattern ①, we have generated two transformed programs ③ (assuming that `a`=`b` and `i`=`j`) and ④ (for the other cases). In the former case, only one ghost variable is needed as `a[i]`≡`b[j]` while, in the latter, as `a[i]`≢`b[j]`, we use g_1 (resp. g_2) to represent `a[i]` (resp. `b[j]`). Standard analysis successfully proves termination of ④, but not ③ since indeed in this case the loop might not terminate. Finally, program ⑤ shows the transformation required to prove termination of the circular array traversal. The main point is that it does not modify the semantics when the precondition holds, while it simplifies the termination proof.

2 Intermediate Representation of Bytecode

We develop our analysis on an intermediate representation (IR) of bytecode [4]. A *program* in the IR consists of a set of *procedure*s which are defined by means of a set of (recursive) *guarded rules*, which adhere to the following grammar:

$$\begin{array}{rl} \mathit{rule} ::= & p(\bar{x},\bar{y}) \ \leftarrow g, b_1, \ldots, b_n \qquad\qquad g ::= \mathit{true} \mid x\ \mathit{op}\ y \\ \mathit{op} ::= & > \mid < \mid \le \mid \ge \mid = \mid \neq \qquad\qquad \mathit{aop} ::= + \mid - \mid * \mid / \\ b ::= & x{:=}\mathit{exp} \mid x[y]{:=}\mathit{exp} \mid q(\bar{x},\bar{y}) \mid \mathtt{assume}(\varphi) \\ \mathit{exp} ::= & x \mid \mathsf{null} \mid n \mid x\ \mathit{aop}\ y \mid \mathsf{newarray}(\mathtt{int}, x) \mid \mathsf{length}(x) \mid x[y] \end{array}$$

where $p(\bar{x}, \bar{y})$ is the *head* of the rule; $\bar{x}$ (resp. $\bar{y}$) are the input (resp. output) parameters; g its guard, which specifies conditions for the rule to be applicable; $b_1, \ldots, b_n$ the body of the rule; n an integer; x and y variables; $x[y]$ the element at position y of an array x, $q(\bar{x}, \bar{y})$ a procedure call by value, $\mathsf{newarray}(\mathtt{int}, x)$ creates an array of x elements of type `int`, $\mathsf{length}(x)$ returns the length of the array x. For the sake of simplicity, we consider only arrays of integers, but the generalization to other types is direct. We adopt the Java Modeling Language (JML) notation $\mathtt{assume}(\varphi)$ to indicate to the analyzer that condition φ can be assumed to hold. A method m in a Java (bytecode) program is represented by a set of procedures in the IR such that there is an entry procedure named m and the remaining ones are intermediate procedures invoked only within m. The translation of a program into the IR works by first building the control flow graph (CFG) from the program, and then representing each block of the CFG in the IR as a rule. The process is identical to [4], hence, we skip the technical details of the transformation and just show the intuition by means of an example.

Example 1. The following IR are obtained from the code of pattern ①:

$$\begin{array}{l|l}
\mathbf{while}(\langle a,b,i,j\rangle,\langle\rangle) \leftarrow & \mathbf{while_c}(\langle a,b,i,j,s_0\rangle,\langle\rangle) \leftarrow s_0 > 0,\\
\quad s_0{:=}a, s_1{:=}i, & \quad s_1{:=}a, s_2{:=}i, s_3{:=}a, s_4{:=}i,\\
\text{ⓐ}\, s_0{:=}s_0[s_1], & \text{ⓐ}\, s_3{:=}s_3[s_4], s_3{:=}s_3-1, s_1[s_2]{:=}s_3,\\
\quad \mathit{while_c}(\langle a,b,i,j,s_0\rangle,\langle\rangle). & \quad s_1{:=}b, s_2{:=}j, s_3{:=}b, s_4{:=}j,\\
\mathbf{while_c}(\langle a,b,i,j,s_0\rangle,\langle\rangle) \leftarrow s_0 \leq 0. & \text{ⓑ}\, s_3{:=}s_3[s_4], s_3{:=}s_3+1, s_1[s_2]{:=}s_3,\\
 & \quad \mathit{while}(\langle a,b,i,j\rangle,\langle\rangle).
\end{array}$$

It receives as input parameters two array references `a` and `b` and two integer values `i` and `j`. The most important point to note is that the accesses to the array are performed by pushing the values to the stack (which in the IR are just local variables). E.g. the first three instructions in the body of procedure `while` push the value of `a[i]` in the stack position s_0.

3 Constancy Analysis for Bytecode

It is essential to use a semantic-based analysis (instead of just performing syntactic checks), because, as we have seen in Ex. 1, at the bytecode level, array elements are not manipulated directly, but rather pushed into the stack. Stack positions are just standard variables at the IR level.

Abstract domain. The analysis is a dataflow analysis which assigns one of the following *symbolic* values to each program variable at every program point:

- $\top$ and $\bot$ represent *any* and *none* information
- null is used to indicate that a variable has the constant value null
- l_i represents the symbolic initial value of the i-th input parameter
- e represents a symbolic arithmetic expression over the input parameters l_i and the integers
- $l_i[e]$ represents a symbolic array reference, where e is a symbolic expression as in the above element or $\top$

We assume that symbolic expressions are given in some normal form. The above values form an abstract domain D, in which the elements are ordered by a relation $\sqsubseteq$ such that $A \sqsubseteq \top$, $\bot \sqsubseteq A$, $l_i[A] \sqsubseteq l_i[\top]$. Using this order, the least upper bound $X \sqcup Y$ is equal to X (resp. Y), if $Y = \bot \vee X = Y$ (resp. $X = \bot$); $l_i[\top]$, if $X = l_i[A] \wedge Y = l_i[B] \wedge A \neq B$; and $\top$ otherwise.

An *abstract state* is a mapping $\phi : V \mapsto D$ where V is the set of variable names. For *integer* variables, the abstract information states whether they keep a *constant* value at this program point and what the value is (in particular, it can be the value stored in an array element, the value of an initial parameter, a constant integer, etc.). For *reference* variables, it states whether the reference remains constant and the symbolic value of such reference. In the analysis, manipulating abstract values can result in abstract values not considered above, namely $\top[A]$ that we consider as $\top$, and $l_i[l_j[A]]$ that we consider as $l_i[\top]$.

Analysis. For simplicity, we present the analysis for a single simple loop P_k with an entry procedure p_k, such that its procedures form a strongly connected component. In what follows we refer to such loop as a scope. The analysis is a standard forward analysis which propagates the initial abstract state, which maps the i-th input parameter to the symbolic value l_i, to the different program points using the following *transfer* function, which describes the effect of executing one instruction on a given abstract state ϕ:

Instruction	*transfer*(b, ϕ)	Instruction	*transfer*(b, ϕ)
(1) $x{:=}n$	$\phi[x \mapsto n]$	(4) $x{:=}\mathsf{newarray}(\mathtt{int}, y)$	$\phi[x \mapsto \top]$
(1) $x{:=}\mathsf{null}$	$\phi[x \mapsto \mathsf{null}]$	(5) $x{:=}y[z]$	$\phi[x \mapsto \phi(y)[\phi(z)]]$
(2) $x{:=}y$	$\phi[x \mapsto \phi(y)]$	(6) $x[y]{:=}exp$	$remove(\phi)$
(3) $x{:=}y\ op\ z$	$\phi[x \mapsto \phi(y)\ \overline{op}\ \phi(z)]$	*otherwise*	ϕ

(1) it modifies ϕ such that x will have the abstract value $n \in \mathbb{Z}$ (resp. null);
(2) it modifies ϕ such that the abstract value of x will be equal to that of y;
(3) it applies (an abstract version of) the arithmetic operator to the abstract values of y and z, and stores the result in x. The operator $\overline{op}$ keeps (a normalized version of) the symbolic expression if both operands are symbolic expressions over integers and l_i, otherwise it is $\top$;
(4) it modifies ϕ such that x will have the value $\top$. This can be refined, if the instruction does not occur in a loop, to assign an abstract value that corresponds to that allocation-site.
(5) it modifies ϕ such that x is mapped to the corresponding array element;
(6) as the array content has been modified, we have to eliminate the information carried for all array elements from ϕ, since they might not be valid anymore. This is done by $remove(\phi)$ which returns ϕ' such that, for all x, $\phi'(x) = \top$ if $\phi(x) = l_i[A]$, and otherwise $\phi'(x) = \phi(x)$;

Example 2. Fig. 2 shows to the left the IR of pattern ② and to the right the abstract states, computed by the analysis, for each program point. The most relevant points are: (1) as variable `i` does not have a constant value during the

$\mathbf{while}(\langle a, x, i\rangle, \langle i\rangle) \leftarrow$	
$s_0{:=}a,$	$\{a \to l_1, x \to l_2, i \to \top\}$
$s_1{:=}i,$	$\{a \to l_1, x \to l_2, i \to \top, s_0 \to l_1\}$
$s_0{:=}s_0[s_1],$	$\{a \to l_1, x \to l_2, i \to \top, s_0 \to l_1, s_1 \to \top\}$
$while_c(\langle a, x, i, s_0\rangle, \langle i\rangle).$	$\{a \to l_1, x \to l_2, i \to \top, s_0 \to l_1[\top], s_1 \to \top\}$
$\mathbf{while_c}(\langle a, x, i, s_0\rangle, \langle i\rangle) \leftarrow$ ⓒ$x = s_0.$	$\{a \to l_1, x \to l_2, i \to \top, s_0 \to l_1[\top]\}$
$\mathbf{while_c}(\langle a, x, i, s_0\rangle, \langle i\rangle) \leftarrow$ ⓒ$x \neq s_0,$	$\{a \to l_1, x \to l_2, i \to \top, s_0 \to l_1[\top]\}$
$s_1{:=}i + 1,$	$\{a \to l_1, x \to l_2, i \to \top, s_0 \to l_1[\top]\}$
$s_2{:=}length(a),$	$\{a \to l_1, x \to l_2, i \to \top, s_0 \to l_1[\top], s_1 \to \top\}$
$i{:=}s_1 \% s_2,$	$\{a \to l_1, x \to l_2, i \to \top, s_0 \to l_1[\top], s_1 \to \top, s_2 \to \top\}$
$while(\langle a, x, i\rangle, \langle i\rangle).$	$\{a \to l_1, x \to l_2, i \to \top, s_0 \to l_1[\top], s_1 \to \top, s_2 \to \top\}$

Fig. 2. IR of Pattern 2 (left). Program point constancy information (right)

loop iterations, its abstract value is $\top$; and (2) we have inferred at the points marked by ⓒ that the stack value s_0 contains the value of an array element l_1`[`$\top$`]`. Importantly, we keep the information that the reference to the array remains constant (even if we lose the array index). Consider again the IR of Pattern ① in Ex. 1, assume that l_1,l_2,l_3,l_4 refer to, resp., the initial values of `a`, `b`, `i` and `j`. Our analysis infers that the array accesses at program points ⓐ are done using the constant access l_1`[`l_3`]`, and at program points ⓑ, using l_2`[`l_4`]`.

One could think of syntactically checking that constancy information instead of developing a semantic-based analysis. However, there exist an enormous variety of loops, ranging from the different shape of those implemented using recursion or iteration, from the number of arrays that they access, the different array positions accessed, etc. Besides, when inferring the information at the level of bytecode (assuming that the source code is not available), it is even more complex due to the use of stack variables as we have seen. Our analysis achieves automation of syntactic checks and it generalizes w.r.t. the structure of the programs (e.g., it allows us to handle iteration and recursion in a uniform way).

4 Automatic Transformation at Bytecode Level

In this section, we define two transformations for *simple* loops which are represented by a single scope (SCC in this case) and without calls to other scopes. In future work, we will extend it to the case of nested loops. First, in Sec. 4.1, we use the information inferred by the analysis of Sec. 3 in order to learn how arrays are accessed. Then, in Sec. 4.2 we instrument the program with additional (ghost) variables which make some properties of the arrays explicit. In turn, these properties become observable by existing termination analysis tools, since now they are properties of local variables rather than of array elements.

4.1 Conditional Trackability

The information on how arrays are accessed in a given scope is obtained from the analysis presented in Sec. 3 as follows.

Definition 1. *For a scope P_k, the multiset of* array read accesses $R(P_k)$ *is* $\{\phi_{ij}(y)[\phi_{ij}(z)] \mid b_{ij} \equiv x{:=}y[z] \in P_k\}$. *The multiset of* array write accesses $W(P_k)$ *is defined similarly by considering write accesses instead of read accesses.*

Intuitively, R includes accesses that are performed in scope P_k. Note that R might contain multiple occurrence of syntactically identical elements. This is important only when two syntactically identical elements involve $\top$, since it is not guaranteed that they are semantically identical, in other cases multiple occurrence can be eliminated.

Example 3. Using the results of the constancy analysis in Ex. 2, for Pattern ②, the resulting read access set contains only one array access $R(P_2) = \{l_1[\top]\}$ while the write access set is empty, $W(P_2) = \emptyset$, because the array content is not updated within the loop. Similarly, the read/write access sets for Pattern ① are $R(P_1) = W(P_1) = \{l_1[l_3], l_2[l_4]\}$ since the array content is read and modified using the references `a[i]` and `b[j]`.

As mentioned before, in some of the transformations, our aim is to simulate some aspects of array elements by replacing the array accesses with local variables. This can be done only if we are able to precisely track the array modifications, i.e., we can definitely tell which array elements are modified. This is not obvious since (1) two local variables might point to the same array – alias; and (2) the index of the accessed element is stored in a variable and thus it is not immediate to know which element we are accessing or if we are always accessing the same element. If the array is not modified, the problem is clearly much simpler.

Example 4. Consider Pattern ① in Fig. 1. We cannot precisely track the write accesses to the arrays (`a` or `b`) because the memory location accessed depends on an aliasing condition: if `a` and `b` point to the same array, the content of such array may be modified using both accesses, `a[i]` or `b[j]`. Furthermore, if $\texttt{i} \equiv \texttt{j}$, both accesses are modifying exactly the same element of the array.

The above example shows that it is often not possible to track array updates unconditionally. Fortunately, since the elements of the read/write sets are given in terms of the initial parameters $l_1, \ldots, l_n$, one could try to provide preconditions on those values such that array accesses become *trackable*.

Example 5. As it is shown in Ex.4, the trackability of array accesses and, as a consequence, the number of ghost variables needed to track them depends on some preconditions that are given in terms of the initial parameters. E.g. assuming the precondition $l_1 \neq l_2 \vee l_3 \neq l_4$ over the read/write sets in Ex. 3, $\{l_1[l_3], l_2[l_4]\}$, we will need two different ghost variables to safely represent these array references, because they are pointing to different memory locations. However, if we assume that $\{l_1 = l_2 \wedge l_3 = l_4\}$, all accesses point to the same array element, so we just need one ghost variable to track both array accesses.

For simplicity, we consider conditions in terms of linear constraints on the symbolic initial values $l_1, ..., l_n$. In principle, any kind of conditions can be used as far as they can be manipulated as in Def. 2. We write $\ell_1{=}\ell_2$ if ℓ_1 and ℓ_2 are guaranteed to refer to the same memory location, and $\ell_1{\neq}\ell_2$ if not.

Definition 2 (conditional trackability). *Given a precondition ψ and a scope for P_k. We say that the array accesses in P_k are trackable w.r.t. ψ if $R(P_k) \cup W(P_k)$ can be partitioned into $G_1 \cup \cdots \cup G_m$ such that: (1) for each G_i we have $\forall \ell_1, \ell_2 \in G_i . \psi \models \ell_1 = \ell_2$; and (2) if $W(P_k) \neq \emptyset$, then for each two different G_i and G_j we have $\forall \ell_1 \in G_i . \forall \ell_2 \in G_j . \psi \models \ell_1 \neq \ell_2$.*

Example 6. Trackability can be checked by considering all possible equalities and disequalities of the elements in $R(P_k) \cup W(P_k)$. Consider the read/write access sets obtained for Pattern ①, in Ex.3, where $R(P_1) = W(P_1) = \{l_1[l_3], l_2[l_4]\}$ and l_1, l_2, l_3, l_4 refer, resp., to the initial value of variables `a`, `b`, `i`, `j`. The following partitions are generated: (1) $G_1 = \{l_1[l_3], l_2[l_4]\}$ which gives us the precondition $\psi_1 = \{l_1{=}l_2 \wedge l_3{=}l_4\}$, or if we refer to the source code variables, then $\psi_1 = \{a{=}b \wedge i{=}j\}$; (2) $G_1{=}\{l_1[l_3]\}$, $G_2{=}\{l_2[l_4]\}$ which gives us the precondition $\psi_2{=}\{l_1{\neq}l_2 \vee l_3{\neq}l_4\}$ and equivalently, using the source code variables, $\psi_2{=}\{a{\neq}b \vee i{\neq}j\}$.

4.2 Transformation for Simple Loops

Pattern #1. The first pattern deals with programs in which arrays are both read and modified in a given scope. For such cases, this transformation is applicable when the array accesses can be tracked, possibly for a given precondition, according to Def. 2. Thus, the first step is to synthesize conditions for which array accesses are trackable, and then we apply this transformation for each one of them. Given a scope P_k, and a precondition ψ such that it partitions the read and write sets into $G_1 \cup \cdots \cup G_m$, the transformation proceeds as follows: (1) it introduces m new ghost variables $g_1, \ldots, g_m$ (one for each partition), and adds them to each rule as input and output parameters; (2) each instruction that reads/writes an array element is replaced with an equivalent one that uses the corresponding ghost variable. This transformation allows tracking the values stored in some arrays by means of local variables. Thus, if the termination argument of loop P_k depends on some array elements, there exists an equivalent argument that uses the ghost variables and that can be observed by existing array-insensitive termination analysis tools. Proving the universal termination of the transformed program, implies the termination of the original w.r.t. ψ.

Example 7. For Pattern ①, using the preconditions and partitions of Ex.6, we generate two versions depicted in the first two columns of Fig. 3. The one in the first column corresponds to ψ_1, and has one ghost variable g_1, and the one in the second column corresponds to ψ_2 and has two ghost variables g_1 and g_2. Observe that the second version always terminates, while the first one might not because both array accesses, `a[i]` and `b[j]`, modify the same array location. Therefore, using the transformed program, we can infer that the original program terminates for the preconditions ψ_2, i.e. $\{a \neq b \vee i \neq j\}$.

Pattern #2. The second pattern deals with programs that use arrays as cyclic data structures. Here, a common operation is to look for a position in the array that is equal to (or different from) a given value x, assuming that the array indeed includes such element. The point is that the program is written in a way

Pattern #1 (pre-cond. ψ_1)	**Pattern #1** (pre-cond. ψ_2)	**Pattern #2**
while$(\langle a,b,i,j,g_1\rangle, \langle g_1\rangle) \leftarrow$ $s_0{:=}a, s_1{:=}i, s_0{:=}g_1,$ $while_c(\langle a,b,i,j,g_1,s_0\rangle,$ $\langle g_1\rangle)$. **while**$_\mathbf{c}(\langle a,b,i,j,g_1,s_0\rangle,$ $\langle g_1\rangle) \leftarrow s_0 \leq 0$. **while**$_\mathbf{c}(\langle a,b,i,j,g_1,s_0\rangle,$ $\langle g_1\rangle) \leftarrow s_0 > 0,$ $s_1{:=}a, s_2{:=}i, s_3{:=}a, s_4{:=}i,$ $s_3{:=}g_1, s_3{:=}s_3-1, g_1{:=}s_3,$ $s_1{:=}b, s_2{:=}j, s_3{:=}b, s_4{:=}j,$ $s_3{:=}g_1, s_3{:=}s_3+1, g_1{:=}s_3,$ $while(\langle a,b,i,j,g_1\rangle, \langle g_1\rangle)$.	**while**$(\langle a,b,i,j,g_1,g_2\rangle,$ $\langle g_1,g_2\rangle) \leftarrow$ $s_0{:=}a, s_1{:=}i, s_0{:=}g_1,$ $while_c(\langle a,b,i,j,g_1,g_2,s_0\rangle,$ $\langle g_1,g_2\rangle)$. **while**$_\mathbf{c}(\langle a,b,i,j,g_1,g_2,s_0\rangle,$ $\langle g_1,g_2\rangle) \leftarrow s_0 \leq 0$. **while**$_\mathbf{c}(\langle a,b,i,j,g_1,g_2,s_0\rangle,$ $\langle g_1,g_2\rangle) \leftarrow s_0 > 0,$ $s_1{:=}a, s_2{:=}i, s_3{:=}a, s_4{:=}i,$ $s_3{:=}g_1, s_3{:=}s_3-1, g_1{:=}s_3,$ $s_1{:=}b, s_2{:=}j, s_3{:=}b, s_4{:=}j,$ $s_3{:=}g_2, s_3{:=}s_3+1, g_2{:=}s_3,$ $while(\langle a,b,i,j,g_1,g_2\rangle,$ $\langle g_1,g_2\rangle)$.	**while**$(\langle a,x,i,g\rangle, \langle i\rangle) \leftarrow$ $s_0{:=}a, s_1{:=}i, s_0{:=}s_0[s_1],$ $while_c(\langle a,x,i,g,s_0\rangle, \langle i\rangle)$. **while**$_\mathbf{c}(\langle a,x,i,g,s_0\rangle, \langle i\rangle) \leftarrow$ $x = s_0$ **while**$_\mathbf{c}(\langle a,x,i,g,s_0\rangle, \langle i\rangle) \leftarrow$ $g = i$. **while**$_\mathbf{c}(\langle a,x,i,g,s_0\rangle, \langle i\rangle) \leftarrow$ $assume(0 \geq g)$, $assume(g > length(a))$, $x \neq s_0$, $s_1{:=}i+1, s_2{:=}length(a)$, $i{:=}s_1 \% s_2$, $while(\langle a,x,i,g\rangle, \langle i\rangle)$.

Fig. 3. Resulting Bytecode after applying the transformations

that uses this assumption, i.e., it would not terminate if the assumption is not satisfied before executing the loop. The challenge in such cases is to automatically synthesize a precondition under which the program terminates.

Consider again Pattern ②, we are looking for an element in the array that is equal to x. Let us assume that we have an integer variable g such that $0 \leq g < a.length \wedge a[g] == x$. Clearly, under this assumption, replacing the condition $x \neq a[i]$ by $x \neq a[i] \wedge i \neq g$ does not change the termination behavior of this loop. The termination of the program with the new condition can be proven using existing termination analysis tools. At the level of the IR, the transformation is applicable when the write set is empty, and proceeds as follows:

1. look for a procedure p, defined by two different rules, such that the guard of one rule is $x = y$ and the guard of the other rule is $x \neq y$;
2. use the result of the analysis of Sec. 3 to verify that x has a constant value (i.e., either it is l_j, an integer n or null), and that y refers to an array access $l_k[\top]$. If this cannot be verified, then the transformation is not applicable;
3. figure out the index i to which $l_k[\top]$ refers. Note that $\top$ here means that the index cannot be expressed as a constant or in terms of $l_1, \ldots, l_n$. However, it can often be expressed using the value of a local variable If this step cannot be carried out, then the transformation is not applicable.
4. add a ghost variable g to all input parameters of all rules in the given scope;
5. replace condition $x \neq y$ by $x \neq y \wedge g \neq i$ and the condition $x = y$ by $x = y \vee g = i$. Add the assumption $\texttt{assume}(0 \leq g < \texttt{length}(a))$ to both rules of procedure p immediately after the condition.

This transformation guarantees that termination of the transformed program implies termination of the original one w.r.t the following precondition: *if the loop entry condition is $x \neq y$ (resp. $x = y$) then the array associated with l_k has an element that is equal to (resp. different from) the constant value of x (i.e., l_j, n or* null*).* Program ⑤ shows the transformation applied to Pattern ②.

Example 8. Program ② in Fig. 1 matches condition in point (1) above. Then, from the results gathered by the constancy analysis in Fig. 2, we can check that `x` has the constant value l_2 and s_0 has the value $l_1[\top]$, thus condition (2) holds. Then, tracking the index of $l_1[\top]$ we conclude that it is i. Therefore, we can safely apply the transformation which results in the program showed in Fig. 3 whose termination is automatically proven (e.g., by COSTA).

5 Conclusions, Related and Future Work

We have outlined the main phases of an analysis to prove termination of loops over arrays. The core of our approach is a *constancy analysis* which tries to infer the (constant) memory locations of array elements. Such analysis has similarities with previous reference constancy analyses (e.g., [2,1]). However, for handling loops over arrays, we need to precisely track array indexes which often involve arithmetic operations (e.g., it is common to access `a[i+1]`). We can then transform those array accesses which are only read or always written from the same reference location within the loop into local variables. The transformed program can be then analyzed by an array-insensitive termination analyzer.

In an extended version of this paper, we plan to formally develop our analysis for the full sequential Java bytecode, investigate its relation to recent techniques for array sensitive analysis [5,6], and prove its correctness. We have concentrated on arrays, but we believe that our results generalize to collections as found in mainstream languages such as C# or Java. In future work, we will attempt to generalize it to collections. We will also carry out a thorough experimental evaluation to assess the efficiency and effectiveness of our approach.

References

1. A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and Inferring Local Non-Aliasing. In *Proc. of PLDI'03*, pages 129–140. ACM, 2003.
2. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Field-Sensitive Value Analysis by Field-Insensitive Analysis. In *Proc. of FM'09*, volume 5850 of *LNCS*, pages 370–386.
3. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Ramírez. From Object Fields to Local Variables: a Practical Approach to Field-Sensitive Analysis. In *Proc. of SAS'10*, volume 6337 of *LNCS*, pages 100–116.
4. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost Analysis of Object-Oriented Bytecode Programs. *Theor. Comput. Sci.*, 413(1):142–159, 2012.
5. P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Proc. of POPL'11*, pages 105–118.
6. P. Cousot, R. Cousot, and F. Logozzo. Precondition inference from intermittent assertions and application to contracts on collections. In *Proc. of VMCAI'11*, volume 6538 of *LNCS*, pages 150–168.
7. C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Termination Analysis of Java Bytecode by Term Rewriting. In Johannes Waldmann, editor, *WST'09*.
8. F. Spoto, F. Mesnard, and É. Payet. A Termination Analyzer for Java Bytecode based on Path-Length. *TOPLAS*, 32(3), 2010.

BCT: A translator from MSIL to Boogie

Michael Barnett and Shaz Qadeer

Microsoft Research
Redmond, WA 98052-6399, USA
{mbarnett,qadeer}@microsoft.com

Abstract. We describe the design and implementation of *BCT*, a translator from Microsoft MSIL into Boogie, a verification language that in turn targets SMT (Satisifiability-Modulo-Theories) solvers. BCT provides a vechicle for converting any program checker for the Boogie programming language into a checker for a language that compiles to Microsoft's .NET platform. BCT is methodology-neutral, precise in encoding the operational semantics of the .NET runtime, and comprehensively covers all features of .NET.

1 Introduction

Static analysis of programs is an increasingly important part of software engineering. Advances both in the theory and implementations of static analysis have made it feasible to apply to real-world programs techniques that had hitherto been confined to toy examples. Analyses based on SMT (Satisfiability-Modulo-Theories) solvers such as Z3 [7] and Yices [8], have become especially popular because of the rich modeling capability, based on first-order logic, provided by such solvers.

Programming languages also have evolved: modern object-oriented managed languages provide a rich semantic environment to which such analyses can be applied. In particular — given *our* environment — we are interested in the static analysis of .NET programs. The .NET platform is a multi-lingual runtime currently supporting popular languages, such as C#, VB, and F#. It is also the platform used for Windows Phone applications. It uses a common intermediate bytecode language, MSIL, [9], which all .NET compilers generate.

In order to use an SMT solver to perform static analysis of a program written in a high-level language, the program (and the property to be verified) are encoded as a first-order logic formula. There is a large semantic gap between such different worlds. The Boogie verification system provides an intermediate language (also called Boogie [12]) that makes it easier to bridge this gap. A Boogie program is simple: it comprises a set of global procedures with a simple type system encompassing boolean and integer values and maps (arrays with an arbitrarily-typed domain). Everything else is encoded as user-defined datatypes, functions, and axioms. The Boogie system generates the input for the solver via a process known as *verification condition generation*.

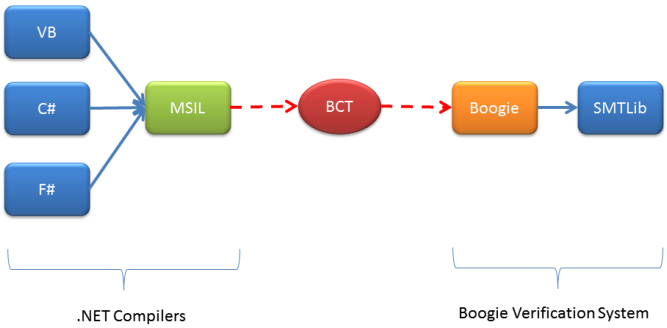


Fig. 1: The translation pipeline showing the translation/compilation steps from high-level (.NET) programming languages to the standard language used by modern automatic theorem provers, SMTLib [5]. Note that **Boogie** refers to the language and not the tool.

While a Boogie program is much closer to one written in a high-level programming language, there is still a gap that must be filled. As shown in Figure 1, we have created a bytecode translator, *BCT*, that translates .NET bytecode, i.e., MSIL, into the Boogie programming language. By using MSIL as our source language, we can immediately apply any particular static analysis to programs written in any .NET language, e.g., C#, VB, and F#.

1.1 Criteria for a translator

There are several essential criteria for a general bytecode translator. Our goal is to have BCT fulfill all of them.

Comprehensive A general bytecode translator should cover all .NET features. It should have a faithful and precise translation of reference and value types, generics, subtyping, delegates, dynamic dispatch and exception handling. Real programs use all of these features so a system that is unable to deal with all of them would not be usable. Currently BCT handles all of these features except for generics; we are currently working on designing a translation for them.

Neutral A general translator should encode only the operational semantics of the .NET runtime. It should not impose any particular verification methodology.

Standard As much as possible, a translator should use common, open, components. Using such standard, reusable, components helps in assuring the correctness of the translator itself. BCT is built on top of the Common Compiler Infrastructure [6], *CCI*, and uses the Boogie Programming Language as

its target language. Not only are both open-source, but CCI uses the .NET de-facto standard encoding for program specifications, Code Contracts for .NET [3]. CCI also provides many needed components, such as recovering boolean expressions from MSIL (which uses integers to encode booleans).

Flexible It is important that a translator is not overly constrained. It should also be possible to modify certain aspects of the translation, e.g., injecting certain assertions to encode a particular verification methodology, with minimal effort, ideally, by supplying a binary component that is called at the right moments during the translation. BCT meets this requirement by providing support for both modular and whole-program analysis. This is especially crucial for programs, such as mobile-phone applications, that make heavy use of *delegates*, type-safe function pointers. A whole-program translation can also encode *dynamic dispatch*, which makes it easier to determine the target of a virtual method call. It also allows different heap representations: we currently support representing it as a two-dimensional map (indexed by object references and field names) or else as a set of one-dimensional maps, one per field (and indexed by object references).

1.2 Related Work

A first version of a bytecode translator [1] from MSIL to Boogie was written as part of the Spec# project [2]. Some of its shortcomings were:

- It used an ad-hoc encoding of *contracts* to persist method pre- and postconditions, object invariants, and non-null type information.
- It did not implement generics, value types, and exception handling.
- It implemented its own dataflow analysis for reconstructing booleans, managed pointers, and type tokens rather than using a common component.
- It was monolithic in that it encoded the Spec# methodology [13, 4] as part of the translation. While many options were added to control aspects of the translation, it was difficult to achieve a methodology-free translation.
- It provided only a modular translation, which limited its ability to handle delegates and dynamic dispatch.

BCT attempts to address all of these limitations of Spec#.

Lehner and Müller [11] describe a sound translation of a subset of Java bytecode to Boogie.

1.3 Applications

There are already several analyses that are taking advantage of BCT in order to analyze .NET programs. Poirot [14] is a static tool for helping programmers detect and debug errors in concurrent and asynchronous systems; it uses BCT as its .NET frontend. The GetMeHere project at Microsoft Research aims to provide a symbolic debugging experience to the programmer, again using BCT for its .NET front end. Both Poirot and GetMeHere use the Corral [10] verifier to solve reachability queries on Boogie programs.

2 The bytecode translator

In this section, we informally explain how BCT translates various features of MSIL. For illustration purposes, we use C# syntax for the source code; however, remember that BCT's input is actually the MSIL generated by the the C# compiler. All types, methods, and fields are implicitly declared as `public`. Although the translator does encode exceptional control flow as well as normal control flow, only the latter is shown.

2.1 Reference types

We begin by explaining our translation for reference types. Figure 2 shows a class `A` with two methods, `Increment` and `IncrementSelf`. The method `Program.Main` allocates an object of type `A` and calls these methods in sequence. Note that allocation is assumed to always succeed. The assertions in the program provide a clue to the expected behavior.

```
class A {
  int f;

  static void Increment(A x) { x.f++; }

  void IncrementSelf() { f++; }

  static void Main() {
    A a = new A();

    Increment(a);
    Contract.Assert(a.f == 1);

    a.IncrementSelf();
    Contract.Assert(a.f == 2);
  }
}
```

Fig. 2: C# code

```
type Ref;
const unique null: Ref;
var $Alloc: [Ref]bool;
var F$A.f: [Ref]int;
procedure {:inline 1} Alloc() returns (x: Ref) {
  assume $Alloc[x] == false && x != null;
  $Alloc[x] := true;
}
procedure A.Increment$A(x$in: Ref) {
  F$A.f[x$in] := F$A.f[x$in] + 1;
}
procedure A.IncrementSelf($this: Ref) {
  F$A.f[$this] := F$A.f[$this] + 1;
}
procedure A.Main() {
  var a_Ref, $tmp1: Ref;
  call $tmp1 := Alloc();
  call A.#default_ctor($tmp1);
  assume $DynamicType($tmp1) == T$A();
  a_Ref := $tmp1;
  call A.Increment$A(a_Ref);
  assert F$A.f[a_Ref] == 1;
  call A.IncrementSelf(a_Ref);
  assert F$A.f[a_Ref] == 2;
}
```

Fig. 3: Boogie code

Figure 3 shows relevant pieces of the Boogie code generated from the bytecode of the program in Figure 3. (For instance, the default constructor for the class `A` is not shown.) We model heap objects using maps, one for each field; for example, field `f` is represented by the global map variable `F$A.f`.

The domain of `F$A.f` is the uninterpreted type `Ref` representing the set of all object references. The range of `F$A.f` is the built-in Boogie type `int` used for representing `System.Int32`, the type of field `f`.

The translation of `A.Main` shows how object allocation is modeled. To ensure that each allocated reference is distinct, BCT uses another map `$Alloc`; each call to the allocator procedure `Alloc` returns a reference `x` such that `Alloc(x)` is false before and true after the call. BCT generates a Boogie procedure representing the default constructor which initializes each field-representing map at `x` to a null-equivalent value. This constructor is called on the reference returned by `Alloc` and assumed to be of the correct dynamic type. Other uses of the dynamic type are shown in Section 2.3. The translation of the code in the `Increment` and `IncrementSelf` procedures is straightforward; one interesting aspect is that the self parameter in `IncrementSelf` is made explicit in the generated Boogie code.

2.2 Value types

In addition to built-in value types such as `bool` and `int`, MSIL allows defining custom value types called structs. The program in Figure 4 illustrates the semantics of struct types. `A` is a struct type with a single field `f`. The static method `Increment` takes a value `x` of type `A` and increments its `f` field. The first three lines of `Main` indicate that struct types are indeed passed by value; the side-effect on `x.f` in `Increment` leaves the value of `a.f` in `Main` unchanged. However, as the next three lines show, the semantics is subtly different when a method `IncrementSelf` is called on the struct value in `a`. In that case, the effect is that a reference to the struct value is passed to the procedure. Thus, a struct behaves like an object when it is a receiver of a method and otherwise behaves as a value.

```
struct A {
  int f;

  static void Increment(A x) { x.f++; }

  void IncrementSelf() { f++; }

  static void Main() {
    A a = new A();
    Increment(a);
    Contract.Assert(a.f == 0);
    a.IncrementSelf();
    Contract.Assert(a.f == 1);
  }
}
```

Fig. 4: C# code

```
procedure A.Main()
{
  var a_Ref: Ref;
  var $tmp1: Ref;
  call $tmp1 := Alloc();
  call A.#default_ctor($tmp1);
  assume $DynamicType($tmp1) == T$A();
  a_Ref := $tmp1;
  call A.#copy_ctor(a_Ref, $tmp2);
  call A.Increment$A($tmp2);
  assert F$A.f[a_Ref] == 0;
  call A.IncrementSelf(a_Ref);
  assert F$A.f[a_Ref] == 1;
}
```

Fig. 5: Boogie code

Just as for fields of reference types, BCT represents struct fields as maps. We model structs as heap-allocated, just like objects. However, the treatment of assignment is different; BCT creates a Boogie procedure modeling a copy constructor for each struct. The copy constructor is used whenever a struct value is assigned: either with an explicit assignment statement or when it is passed as an argument to a method, as in `A.Increment`.

2.3 Subtyping and dynamic dispatch

Figure 6 shows a program containing an abstract class `A` with two subclasses `B` and `C`. We use this example to show how BCT translates calls to virtual methods. BCT explicitly maintains the dynamic type of each allocated object using the function `$DynamicType`. (We use a function instead of a map since this information is not mutable.) The type itself is represented at the Boogie level as an abstract datatype with one constructor per declared type in the program. In Figure 7, we see the declaration of the abstract datatype `Type` and its three constructors—`T$A`, `T$B`, and `T$C`. Each constructor is associated with a membership testing function on elements of `Type`. For example, `is#T$C(t)` returns true iff `t` is equal to `T$C()`. The subtype relation on types is captured by the `$Subtype` predicate; we omit the axiomatization of this predicate from this paper for space reasons.

```
abstract class A {
  int f,g;
  abstract void Increment();
}

class B : A {
  override void Increment() { f++; }
}

class C : A {
  override void Increment() { g++; }
}

class Program {
  static void Increment(A a) {
    Contract.Assert(a is B || a is C);
    a.Increment();
  }

  static void Main() {
    var b = new B();
    Increment(b);
    Contract.Assert(b.g == 0);
    var c = new C();
    Increment(c);
    Contract.Assert(c.f == 0);
  }
}
```

Fig. 6: C# code

```
type {:datatype} Type;
function {:constructor} T$A() : Type;
function {:constructor} T$B() : Type;
function {:constructor} T$C() : Type;

function $DynamicType(Ref) : Type;
function $Subtype(Type, Type) : bool;
procedure Program.Increment$A(a$in: Ref)
{
  var a: Ref;
  a := a$in;
  assert(a != null && $Subtype($DynamicType(a), T$B()) ||
         a != null && $Subtype($DynamicType(a), T$C()));

  if (is#T$C($DynamicType(a)))
  {
    call C.Increment(a);
  }
  else if (is#T$B($DynamicType(a)))
  {
    call B.Increment(a);
  }
  else
  {
    call A.Increment(a);
  }
}
```

Fig. 7: Boogie code

The translation of the method `Program.Increment` shows how the `$Subtype` predicate is used to translate both conditionals related to type queries and virtual calls. A virtual call is translated by testing for the dynamic type of the reference and dispatch to the appropriate procedure based on the result. This is dependent on doing a whole-program translation: a modular translation would contain a call only to the static type of the receiver.

2.4 Delegates

In MSIL, a delegate is a type-safe reference to a method; when the method is an instance method, the delegate also contains a reference to the object on which the method should be called. BCT uses algebraic datatypes to encode the value of a delegate as a pair, as shown in Figure 9. Each method name is modeled as a unique integer constant. `$DelegateCons` is the constructor and `$Method#DelegateCons` and `$Receiver#DelegateCons` are the selectors that return the method and the receiver object respectively. The translation of the `TakeAction` method models the delegate call inside it with a call to the Boogie procedure `Program.Action.Invoke`. This makes explicit the delegate dispatch. The body of `Program.Action.Invoke` retrieves the method name from the delegate value and dispatches to the appropriate Boogie procedure. Note that our translation of delegates depends on a whole-program analysis that calculates the set of all methods that could potentially be present in a delegate variable of a particular type.

```
class O {
  int f;
  void Set() { f = 42; }
}

class Program {
  delegate void Action();

  static bool flag = false;
  static void SetFlag() {
    flag = true;
  }
  static void TakeAction(Action a) { a(); }

  static void Main() {
    TakeAction(SetFlag);
    Contract.Assert(flag);
    O o = new O();
    TakeAction(o.Set);
    Contract.Assert(o.f == 42);
  }
}
```

Fig. 8: C# code

```
type {:datatype} Delegate;
function {:constructor}
$DelegateCons($Method: int, $Receiver: Ref) : Delegate;

const unique O.Set: int;
const unique Program.SetFlag: int;

procedure Program.Action.Invoke(delegate: Delegate)
{
  var method: int;
  var receiver: Ref;

  method := $Method#$DelegateCons(delegate);
  receiver := $Receiver#$DelegateCons(delegate);
  if (method == O.Set) {
    call O.Set(receiver);
  } else if (method == Program.SetFlag) {
    call Program.SetFlag();
  } else if (true) {
    assume false;
  }
}
```

Fig. 9: Boogie code

3 Conclusions

Space limitations prevent us from showing many other features of the translator. In addition to exceptions (mentioned in Section 2), we have left out the translation for .NET's concurrency features. There are several places where BCT is currently *incomplete*: it ignores unsafe or unmanaged code, reflection, object invariants, and modifies clauses.

We are continuing to work on the translator. We would like to finish the design and implementation of our generics translation. We are also designing a plugin model to support domain-specific methodologies and program instrumentations. We would also like to explore the translator's use to verify code written using Code Contracts. Since the project is open-source (as part of the Boogie CodePlex site), we look forward to collaborating with others on it.

References

1. Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, September 2006.
2. Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and verification: The Spec# experience. *Communications of the ACM*, 54(6):81–91, June 2011.
3. Mike Barnett, Manuel Fähndrich, and Francesco Logozzo. Embedded contract languages. In *ACM SAC - OOPS*. ACM, March 2010.
4. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2005.
5. Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, 2010.
6. The common compiler infrastructure (CCI). http://ccisamples.codeplex.com/.
7. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer-Verlag, 2008.
8. Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for dpll(t). In *CAV*, pages 81–94, 2006.
9. ECMA. Standard ECMA-355, Common Language Infrastructure, June 2006.
10. Akash Lal, Shaz Qadeer, and Shuvendu Lahiri. Corral: A whole-program analyzer for boogie. Technical Report MSR-TR-2011-60, Microsoft Research, May 2011.
11. H. Lehner and P. Müller. Formal Translation of Bytecode into BoogiePL. In M. Huisman and F. Spoto, editors, *Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)*, volume 190(1) of *Electronic Notes in Theoretical Computer Science*, pages 35–50, 2007.
12. K. Rustan M. Leino. This is Boogie 2. Manuscript KRML 178, 2008. Available at http://research.microsoft.com/ leino/papers.html.
13. K. Rustan M. Leino and Peter Müller. Spec# tutorial web page, 2009. http://specsharp.codeplex.com/Wiki/View.aspx?title=Tutorial.
14. Poirot: a concurrency sleuth. http://research.microsoft.com/en-us/projects/poirot/.

Log-based Lazy Monitoring of OSGi Bundles*

Giulio Caravagna[1], Gabriele Costa[2**], Giovanni Pardini[3], and Luca Wiegand[2]

[1] Dipartimento di Informatica, Sistemistica e Comunicazione,
Università degli Studi di Milano-Bicocca, Italy
giulio.caravagna@disco.unimib.it
[2] Institute for Informatics and Telematics,
National Research Council, Pisa, Italy
{gabriele.costa,luca.wiegand}@iit.cnr.it
[3] Dipartimento di Informatica,
Università degli Studi di Verona, Italy
giovanni.pardini@univr.it

Abstract. *Lazy controllers* are execution monitors which do not continuously observe the behaviour of their target. Monitors are activated and deactivated according to a scheduling strategy. When a lazy controller is activated, it checks the current security state and, in case of a violation, terminates the execution. Otherwise, if the current execution trace is safe, the monitor is suspended and its activation is scheduled again. The inactivity period is computed by considering the risk that, from the current state, the target can produce a security violation.
In this paper we present a prototype using existing logging API, i.e., the *Commons Logging Package*, for remotely watching the execution of OSGi bundles. We claim that our solution can efficiently follow the target system keeping under control the delay in detecting violations. Also, as we use standard OSGi platform and facilities, we show that our monitors can run under very realistic assumptions for bundle applications.

1 Introduction

Security monitors are commonly used for controlling that a target respects some security requirements. Many authors proposed important contributions to the theory and practice of security controllers, e.g., see [1,2,3,4,5]. All these approaches present security controllers that can guard program executions and run reaction procedures. Several other proposals, e.g., see [6,7,8,9], also exploit a verification step for supporting the synthesis and execution of security monitors.

Many of the most influential proposals for modelling security controllers, e.g., see [1] and [10], rely on a continuous, step by step observation of the target execution. Although this is a reasonable approach when monitoring the local

* Work partially supported by EU-funded projects FP7-231167 Connect, FP7-257930 Aniketos, FP7-256980 NESSoS and by FP7-257876 SPaCIoS.

** Now at Dipartimento di Informatica, Sistemistica e Telematica, Univeristà di Genova, Italy.

execution of a program, it may be difficult or even impossible to implement the same strategy when the target is expected to execute remotely, e.g., consider the Software as a Service (SaaS) paradigm. Also, application monitoring usually requires some modification to either the target program, e.g., code instrumentation, or the execution platform, e.g., system calls wrapping. These techniques do not fit with the remote execution scenario where the mobile code is digitally signed and the execution platform must comply with standard specifications.

For these reasons, we presented a new class of security controllers, namely *lazy controllers* [11]. Like standard controllers, lazy controllers watch their target execution. However, unlike standard monitors, they can autonomously decide to suspend the observations for a certain time span. Clearly, in this way, a lazy controller could miss the observation of a security violation while it is suspended.

Hence, a crucial aspect of the applicability of lazy controllers is the definition and the calculation of the "risk" deriving from pausing the controller guarding the target. A good scheduling for the observations can prevent unobserved security violations, but there is no general, non-trivial way of finding such a scheduling. Intuitively, minimizing the possibility of having a bad scheduling is the main issue when using lazy controllers.

Being able to asynchronously control the target activity has some advantages. In terms of performance and costs, for instance, the monitoring process can be optimised by reducing the number of validity checks on the target behaviour. Another important advantage is in terms of applicability. Indeed, our controllers can be implemented by using existing facilities, while most other approaches use ad-hoc solutions as discussed above. For instance, log auditing [12,13] is often used to asynchronously check the last actions performed by a system.

In this paper we present an implementation of our lazy controllers for the execution monitoring of Java OSGi bundles [14]. Intuitively, we remotely monitor the execution of a bundle by inspecting its execution log. We assume bundles to use the Common Logging API [15] for writing their execution log. Then, we execute the lazy monitor on a different platform. The lazy monitor can request to the bundle execution platform an instance of its log, i.e., a plain sequence of security operations performed by the bundle. When a violation is discovered, the monitor changes the status of its target from *active* to *stopped*.

We show that our method offers substantial advantages w.r.t. a standard security monitor applying the same security policy. These advantages are mainly in terms of performances, i.e., we produce a significantly lower overhead on the system, and applicability, i.e., we can use our approach under very realistic assumptions. As a matter of fact, every OSGi platform provides some logging facilities to installed bundles.

The paper is structured as follows. In Section 2 we briefly introduce lazy controllers and their features. In Section 3 we discuss our prototype implementation and its behaviour and Section 4 concludes the paper.

$$\texttt{(Sleep)}\ \frac{\zeta(C,h)=k \quad k>0}{\langle t, [\![C]\!]_0 \triangleright \{\!|S|\!\}_h\rangle \stackrel{\cdot}{\longrightarrow}_{\mathrm{lzy}} \langle t, [\![C]\!]_k \triangleright \{\!|S|\!\}_h\rangle}$$

$$\texttt{(Mon)}\ \frac{\zeta(C,h)=0 \quad t'=t-h \quad \langle t', C\triangleright S\rangle \stackrel{\alpha}{\Longrightarrow} \langle t'+x, C'\triangleright S'\rangle \quad h\leq x}{\langle t, [\![C]\!]_0 \triangleright \{\!|S|\!\}_h\rangle \stackrel{\alpha}{\longrightarrow}_{\mathrm{lzy}} \langle t'+x, [\![C']\!]_0 \triangleright \{\!|S'|\!\}_0\rangle}$$

$$\texttt{(Log)}\ \frac{\langle t-h, S\rangle \stackrel{a}{\longrightarrow}_{\mathrm{sys}} \langle t+h', S'\rangle \quad h'=x-h \quad h\leq x<h+k \quad C \stackrel{\tilde{a}}{\longrightarrow}_{\mathrm{up}} C' \quad k'=k-h'}{\langle t, [\![C]\!]_k \triangleright \{\!|S|\!\}_h\rangle \stackrel{a}{\longrightarrow}_{\mathrm{lzy}} \langle t+h', [\![C']\!]_{k'} \triangleright \{\!|S'|\!\}_0\rangle}$$

$$\texttt{(Wake)}\ \frac{k>0 \quad h'=h+k \quad t'=t+k}{\langle t, [\![C]\!]_k \triangleright \{\!|S|\!\}_h\rangle \stackrel{\cdot}{\longrightarrow}_{\mathrm{lzy}} \langle t', [\![C]\!]_0 \triangleright \{\!|S|\!\}_{h'}\rangle}$$

Fig. 1: The transition relation $\rightarrow_{\mathrm{lzy}} \subseteq \mathbb{D} \times \mathbb{T} \times \mathbb{D}$.

2 Lazy Security Controllers

We briefly recall the theory of lazy security controllers [11]. A security monitor is a tuple $(\Sigma, \mathcal{C}, \Longrightarrow)$ where $\mathcal{C}$ is a set of states and $\Longrightarrow$ is a transition relation triggered by Σ actions. We write $C \triangleright S \stackrel{\alpha}{\Longrightarrow} C' \triangleright S'$ to denote that the composition of a system in state $S \in \mathcal{S}$, where $\mathcal{S}$ is the set of states of the target, with a controller in state C performs a visible action α. The new system and controller states are S' and C', respectively. Given a discrete/continuous time domain $\mathbb{T}$ a lazy controller is defined as follows.

Definition 1 (Lazy Controller). *A* lazy controller *is* $(\Sigma, \mathcal{C}, \Longrightarrow, \rightarrow_{\mathrm{up}}, \zeta)$ *where:*

- $\Longrightarrow \subseteq (\mathbb{T} \times \mathcal{C} \times \mathcal{S}) \times (\Sigma \cup \{\cdot\}) \times (\mathbb{T} \times \mathcal{C} \times \mathcal{S})$ *is the* active monitoring relation*;*
- $\rightarrow_{\mathrm{up}} \subseteq \mathcal{C} \times \widetilde{\Sigma} \times \mathcal{C}$ *is the* update relation *for unseen actions;*
- $\zeta : \mathcal{C} \times \mathbb{T} \rightarrow \mathbb{T}$ *is the* scheduling function.

Where $\widetilde{\Sigma} = \{\tilde{a} \mid a \in \Sigma\}$ *is the set of unseen actions.*

Relation $\rightarrow_{\mathrm{up}}$ is the operational notion of *activity logging*: while the controller is not observing the system, i.e., it is *idle*, every action $a \in \Sigma$ performed by the target is logged as unseen, i.e., it is $\tilde{a}$ and is freely performed by the target. Instead, function ζ is used to schedule the observations over the execution of the target. We assume that the controller states and time allow for the evaluation of such a sensitive information. In [11] the synthesis of lazy controllers for non-deterministic timed systems with non-instantaneous actions and for both discrete-time and continuous-time markovian probabilistic systems is discussed. In each case the (analytical) probability that a lazy controller misses the detection of a violation is given.

Let $\mathbb{D}$ be the set of all the configurations of the form $\mathbb{T} \times \mathcal{C} \times \mathbb{T} \times \mathcal{S} \times \mathbb{T}$ and let $\mathbb{A} = \Sigma \cup \widetilde{\Sigma} \cup \{\cdot\}$. A Labelled Transition System (LTS) is a graph with

states and labelled edges between states. States denote system configurations and edges transitions from a configuration to another. The semantics of a lazy controller is a LTS $(\mathbb{D}, \mathbb{A}, \rightarrow_{\text{lzy}})$ where $\mathbb{D}$ is the set of states, $\mathbb{A}$ is the set of labels and $\rightarrow_{\text{lzy}} \subseteq \mathbb{D} \times \mathbb{A} \times \mathbb{D}$ is the least transition relation defined by the inference rules of Figure 1. The rules are given in the form $\frac{\text{premises}}{\text{conclusion}}$, along the lines of the Structural Operational Semantics approach [16]; here we informally comment on the rules (see Fig. 1) and we refer the interested reader to [11].

- `(Sleep)` states that, if at time t the controller is active, denoted by $[\![C]\!]_0$, and the next observation is scheduled at time $t + k$, then the controller can idle till that time, here denoted as $[\![C]\!]_k$. The transition label $\cdot$ means that this derivation does not involve any action of the target
- `(Mon)` applies when the controller is actively following the target. As the scheduler prevents the monitoring from becoming idle, i.e., $\zeta(C, h) = 0$, any action of the target started at $t-h$ and completing at $t-h+x$ is monitored. Relation $\Longrightarrow$ characterizes this behaviour of the controller;
- `(Log)` states that, if the time is t and the controller has scheduled the next observation at time $t+k$, then any action which the target S performs before $t + k$ is not controlled, but simply logged by means of the derivations using $\rightarrow_{\text{up}}$ transition. In this time-window a violation may happen, not being detected up to time $t + k$;
- `(Wake)` allows the controller to spend time autonomously and synchronously with the target S.

Lazy controllers include standard security controllers at the semantic level, as stated in the following theorem whose proof can be found in [11].

Theorem 1 ([11]). *Let $(\Sigma, \mathcal{C}, \Longrightarrow)$ be a security controller, let $(\Sigma, \mathcal{C}, \Longrightarrow, \rightarrow_{\text{up}}, \zeta)$ be the lazy security controller with $\rightarrow_{\text{up}}$ arbitrarily defined and $\zeta : \mathcal{C} \times \mathbb{T} \rightarrow \{0\}$. Then, for any target $S \in \mathcal{S}$ and time $t \in \mathbb{T}$*

$$\langle t, C \triangleright S\rangle \overset{\omega}{\Longrightarrow}{}^{*} \langle t', C' \triangleright S'\rangle \quad \textit{iff} \quad \langle t, [\![C]\!]_0 \triangleright \{\!|S|\!\}_0\rangle \overset{\omega}{\longrightarrow}{}^{*}_{\text{lzy}} \langle t', [\![C']\!]_0 \triangleright \{\!|S'|\!\}_0\rangle\,.$$

In words, Theorem 1 says that, forcing a lazy controller to be always active, i.e. $\zeta(C, t) = 0$ for any C and t, we obtain the same enforcement process produced by the corresponding security controller.

3 Prototype implementation and discussion

In this section we present our prototype and we discuss its behaviour and performance. In order to run our prototype under realistic settings, we defined a case study in which a service running on a remote OSGi platform is monitored.

Case study. We consider a simple medical prescription service infrastructure. The system consists of four actors: (i) a prescription service, (ii) doctors, (iii) pharmacies and (iv) a delivery service. Registered doctors can use the prescription service to fill prescription forms for their patients and submit them to a pharmacy or to the delivery service. Briefly, the program works as follows:

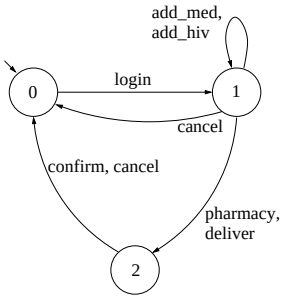


Fig. 2: The prescription system FSM.

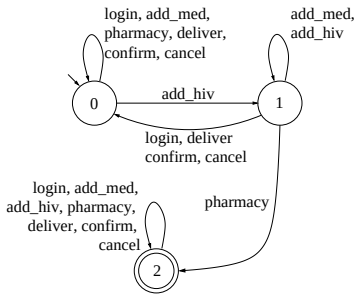


Fig. 3: The privacy policy FSM.

1. initially, the system waits for users, i.e., doctors, to log in (`login`);
2. then the doctor can add one or more medicines (standard, i.e. `add_med`, or HIV-specific, i.e., `add_hiv`) to the prescription;
3. finally, the doctor chooses between two modalities, either `pharmacy` or `deliver`, which specify how the patient gets access to the medicines.

At each step, the doctor can cancel (`cancel`) the operation and, at the end, he must confirm (`confirm`) the prescription. Figure 2 shows the finite state machine (FSM) representing the prescription system.

In order to avoid privacy violations, HIV therapies must always be delivered at the patient's residence. The FSM of Figure 3 represents the privacy policy described above. Briefly, the policy reaches the final state, i.e., detects a violation, if a session in which `add_hiv` has been invoked concludes with `pharmacy`.

Prototype structure. The OSGi bundle implementing the prescription service mainly consists of a simple RMI interface. The interface declares a method for each action labelling the FSM of Figure 2, e.g., `deliver`() for `deliver`. Each method behaves according to its specification, e.g., `add_med`() adds a medicine to the current prescription, and writes a new entry in the log. Logging functionalities are provided by an implementation of the `org.apache.commons.logging.Log` interface that simply appends the given label and a timestamp to a text file.

The lazy controller is an external application, i.e., running on a different platform w.r.t. the target service. At each control cycle, the monitor wakes up and requests the (fragment since the last request of the) current log to the remote platform. Then, the log trace is processed by the policy automaton in Fig. 3 to check if a violation has occurred. In case of a violation, the monitor sends a security error signal to the execution platform (here causing the target to be reinitialised). On the other hand, if the observed trace is legal, the lazy monitor just schedules the next control cycle and hibernates.

The scheduling function maps a pair of states p, for the target, and q, for the policy, into a hibernation time $t_{p,q} \in \mathbb{R}^+$. We compute hibernation times before starting the monitoring process. In this way, we carry out the computation only once and we store the pairs $\langle (p, q), t_{p,q} \rangle$ in a two-column table. Hibernation times are computed using the procedure detailed in [11], starting from a

description of the target system. Clearly, the behaviour of the system depends on the users/doctors. We assume that standard behaviour is known, e.g., by analysing the system execution. In our model we used two different descriptions: Continuous Time (CTMC) and Discrete Time Markov Chains (DTMC). In particular, we use the following matrices for describing the standard execution of the service:

$$R = \begin{bmatrix} 0 & 1/30 & 0 \\ 1 & 2/5 & 1/4 \\ 2 & 0 & 0 \end{bmatrix} \quad P = \begin{bmatrix} 0 & 1 & 0 \\ 1/20 & 17/20 & 1/10 \\ 0 & 0 & 1 \end{bmatrix}$$

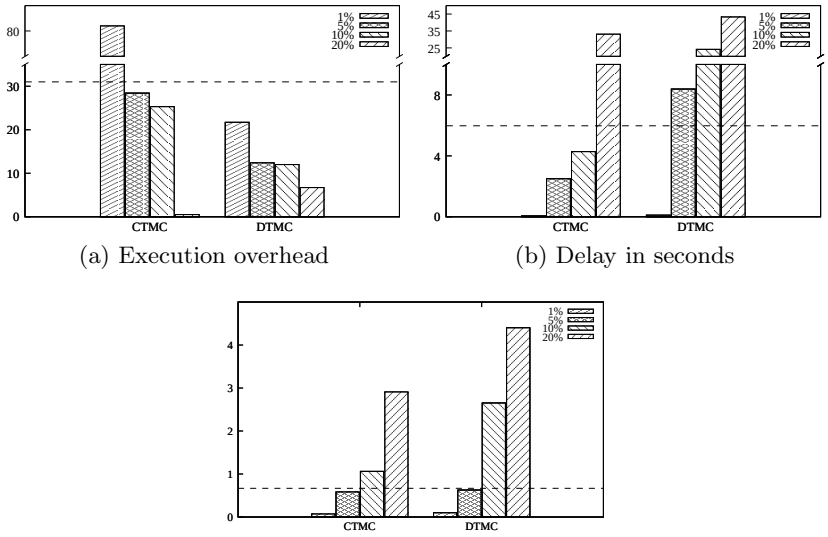
The matrices describe the expected behaviour of the FSM of Figure 2. Matrix R contains rates of state transitions, corresponding to the parameters of exponentially distributed random variables, while P contains the probabilities of state transitions. Intuitively, time rates define the expected number of state transitions per second, e.g., $R[1,2] = 1/30$ means that a transition from state 0 to state 1 happens, on the average, every 30 seconds. Instead, the elements of P describe the probability of moving from the current state to the next one, e.g., $P[2,3] = 1/10$ means that state 3 has $1/10$ probability to be the successor of 2. Also, note that R and P can collapse the values for more than one transition in a single value, e.g., $P[2,2] = 17/20$ denotes both `add_med` ($\mathcal{P}_{\texttt{add\_med}} = 4/5$) and `add_hiv` ($\mathcal{P}_{\texttt{add\_hiv}} = 1/20$) transitions.

Performance evaluation. The prescription service was developed with Eclipse Helios SR2 and executed on OSGi platform Equinox 3.3. Log libraries have been developed implementing the Apache Commons Logging API version 1.1.1.

We tested our system by automatically generating customer sessions of several types. Customers access the system which is monitored using a lazy controller. We synthesize the lazy controllers using the two matrices R and P introduced above and considering four different risk factors, i.e., 0.01, 0.05, 0.1 and 0.2. Also, we compared our monitors with a lazy controller which uses a scheduling function that returns the duration of the shortest path leading to a violation from the current state, computed by means of the Dijkstra algorithm (note that the relation between this scheduling function and the notion of "risk" is not defined in general and, also, the scheduling times cannot be refined). For this purpose, we considered the matrix R' such that $R'[i,j] = R[i,j]^{-1}$ (and $R'[i,j] = \infty$ if $R[i,j] = 0$).

For the overhead analysis we considered customers that statistically behave in a compliant way with respect to the original specification, i.e., the behavioural matrices. The execution overhead is a measure of the computational effort due to the monitoring activity in comparison with the computation of the target. For the continuous time model we considered the activity time of the monitor against the overall execution interval. Instead, for the discrete time model we compared the number of controller synchronizations and the total number of service invocations. Figure 4a shows the simulation output.

As expected, both the approaches increase their performance with the growth of the risk threshold. Moreover, in general they perform better than the Dijk-



(a) Execution overhead

(b) Delay in seconds

(c) Delay in steps

Fig. 4: Monitoring performance evaluation.

stra algorithm-based solution (dashed line). Clearly, such version does not gain advantage from the risk modification.

In order to test delays in violations detection, we executed our system with clients that only emit illegal traces (in the sense of Figure 3). The violating traces are generated using the same probabilities and rates of standard clients. Figures 4b and 4c show the violation detection delays observed in our tests.

Note that the delays for CTMC and DTMC-based monitors have completely different meanings and must be interpreted. Indeed, CTMC controllers work under real time settings, i.e., the monitor is created for keeping under control the time delay of a violation detection rather than the number of actions. Conversely, DTMC controllers aim at minimising the number of actions after a violation. However, it is interesting to compare the two models in both cases.

Finally, we also introduced an error factor for testing the stability of our solution. In particular, we considered users that do not perfectly comply with the given specifications, i.e. the matrices R and P. Interestingly, we found that the performance and delay of our system are stable even with errors up to 30%.

4 Conclusion

We presented a prototype implementation of a monitoring environment using lazy controllers for verifying the policy compliance of a remote execution. Our technique schedules security checks during its execution rather than controlling the target continuously. Although this generates a risk factor, it also extends the applicability of security monitors to many real-world scenarios. Moreover,

we have shown that the risk of a security violation can be analysed and kept under control through the execution parameters of the controllers.

Lazy controllers are generated starting from the specification of a standard security controller. Then we add time constraints to the application rules. In this way, we can convert any existing security controller to a corresponding lazy one. This amount to say that we can apply our solution to existing enforcement environments without redesigning them.

Finally, we considered the performances of our prototype under several settings and we showed execution statistics. The prototype was executed with realistic setting and applied to a case study using OSGi technology. All the experiments highlighted the flexibility and efficiency of our solution.

References

1. Schneider, F.B.: Enforceable security policies. ACM Transactions on Information and System Security **3** (2000) 30–50
2. Bauer, L., Ligatti, J., Walker, D.: More enforceable security policies. In: Foundations of Computer Security. (2002)
3. Colcombet, T., Fradet, P.: Enforcing trace properties by program transformation. In: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '00, New York, NY, USA, ACM (2000) 54–66
4. Thiemann, P.: Enforcing Safety Properties Using Type Specialization. In: Proceedings of the 10th European Symposium on Programming Languages and Systems. ESOP '01, London, UK, Springer-Verlag (2001) 62–76
5. Maler, O., Nickovic, D., Pnueli, A.: From MITL to Timed Automata. In: 4th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS). Volume 4202 of Lecture Notes in Computer Science., Springer (2006) 274–289
6. Skalka, C., Smith, S.: Static enforcement of security with types. SIGPLAN Notices **35** (2000) 34–45
7. Bartoletti, M., Degano, P., Ferrari, G.L., Zunino, R.: Types and effects for resource usage analysis. In: 10th International Conference on Foundations of Software Science and Computational Structures (FOSSACS). Lecture Notes in Computer Science, Springer (2007) 32–47
8. Dragoni, N., Massacci, F., Naliuka, K., Siahaan, I.: Security-by-contract: Toward a semantics for digital signatures on mobile code. In: 4th European PKI Workshop: Theory and Practice (EuroPKI). Volume 4582 of Lecture Notes in Computer Science., Springer (2007) 297–312
9. Martinelli, F., Matteucci, I.: Through modeling to synthesis of security automata. Electronic Notes in Theoretical Computer Science **179** (2007) 31–46
10. Ligatti, J., Bauer, L., Walker, D.: Edit automata: enforcement mechanisms for run-time security policies. International Journal of Information Security **4** (2005) 2–16
11. Caravagna, G., Costa, G., Pardini, G.: Lazy Security Controllers. Technical Report IIT TR-28/2011, Istituto di Informatica e Telematica, Consiglio Nazionale delle Ricerche (2011) `http://puma.isti.cnr.it/`.
12. Garfinkel, S., Spafford, G.: Practical Unix and Internet security (2nd edition). O'Reilly & Associates, Inc., Sebastopol, CA, USA (1996)

13. Axelsson, S., Lindqvist, U., Gustafson, U., Jonsson, E.: An Approach to UNIX Security Logging. In: Proceedings of the 21st NIST-NCSC National Information Systems Security Conference. (1998) 62–75
14. The OSGi Alliance: OSGi Service Platform Core Specification, Release 4, Version 4.3. http://www.osgi.org/ (2011)
15. The Apache Software Foundation: Common Logging API Technical Guide. http://commons.apache.org/logging/tech.html (2011)
16. Plotkin, G.: The Origins of Structural Operational Semantics. In: Journal of Logic and Algebraic Programming. 60-61 (2004) 3–15

Extended Abstract: Embeddable Security-by-Contract Verifier for Java Card*

Olga Gadyatskaya, Eduardo Lostal, and Fabio Massacci

DISI, University of Trento,
via Sommarive, 14, Povo 0, Trento, Italy, 38123
{surname} @disi.unitn.it

Abstract. Modern multi-application smart cards based on the Java Card technology can become an integrated environment where applications from different providers are loaded on the fly and collaborate in order to facilitate lives of the cardholders. This initiative requires an embedded verification mechanism to ensure that all applications on the card respect the application interactions policy.
The Security-by-Contract (S×C) approach for loading time verification consists of two phases. During the first phase the loaded bytecode is verified to be compliant with the supplied contract. Then, during the second phase the contract is matched with the smart card security policy. In the paper we report about implementation of a S×C prototype, present the memory statistics that justifies the potential of this prototype to be embedded on an actual device and discuss the Developer S×C prototype that can be run on a PC.

1 Introduction

Multi-application smart cards are an appealing business scenario for both smart card vendors and smart card holders. Applications interacting on such cards can share sensitive data and collaborate, while the access to the data is protected by the tamper-resistant integrated circuit environment. In order to enable such cards a security mechanism is needed which can ensure that policies of each application provider are satisfied on the card. Though a lot of proposals for access control and information flow policies enforcement for smart cards exist [2, 7, 9], they fall short when the cards can evolve after issuance. The scenario of a dynamic and unexpected post-issuance evolution of a card, when applications from potentially unknown providers can be loaded or removed, is very novel.

For a dynamic scenario, traditionally, run-time monitoring is the preferred solution. But smart cards do not have enough computational capabilities for implementing complex run-time checks. Thus the proposal to adapt the Security-by-Contract approach (initially developed for mobile devices [3]) for smart cards appeared. In the Security-by-Contract (S×C) approach each application supplies

* This paper is a short version of [5]. It provides the high-level engineering aspects of the research results.

on the card its contract, which is a formal description of the application behavior. The contract is verified to be compliant with the application code, and then the system can ensure that the contract matches the security policy of the card.

The S×C framework deployed on the card ensures that all the loaded applications interact in compliance with the security policy of each application provider. In comparison with the existing works aiming at enforcing application interaction policies in a dynamic setting [4, 6], we improve the state of the art in the following (1) the S×C prototype was implemented to be integrated with an actual device, taking into account the memory usage restrictions, (2) we have developed the full eco-system of the S×C verifier based on the standard Java Card tools and specifications available, (3) we have implemented also a version for developers that can be run on a Windows-based PC.

The rest of the paper is structured is follows. Section 2 contains a brief overview of the Java Card technology and then we outline the S×C solution for Java Card (Section 3) emphasizing the changes to the platform. The design and implementation details are outlined in Section 4 For on-card prototypes small memory footprint is a must, we therefore present the memory usage statistics (for non-volatile memory and RAM) that demonstrates feasibility of the embedded implementation (Section 5). We conclude with Section 6.

2 The Java Card Platform

Java Card is a popular middleware for multi-application smart cards that allows post-issuance installation and deletion of applications. Application providers develop *applets* (Java Card applications) in a subset of Java. Full description of the Java Card language is provided in the official specifications [8]. Figure 1 presents the architecture of an integrated circuit with the Java Card platform installed and the application loading process. The architecture comprises several layers including device hardware, an embedded operating system (native OS), the Java Card run-time environment (JCRE) and the applications installed on top of it. Important parts of the JCRE are the Java Card virtual machine (JCVM) (its Interpreter part), the Installer, which is an entity responsible for post-issuance installation and removal of applications and the Loader, that comprises a set of API to access the loaded bytecode.

Applications are supplied on the card in packages. The source code of a package is converted by the application providers into class files and then into a CAP file. The CAP file is, essentially, an optimized Java Card bytecode, it consists of several efficiently organized components each carrying specific information. The CAP file is transmitted onto a smart card, where it is processed and linked.

The interactions between applets from different packages are mediated by the JCRE firewall. If two applets belong to different packages, their *contexts* are different, and the Java Card firewall confines applet's actions to its designated context. Thus, normally, an applet can reach only objects belonging to its own context. The only applet's objects accessible through the firewall are methods

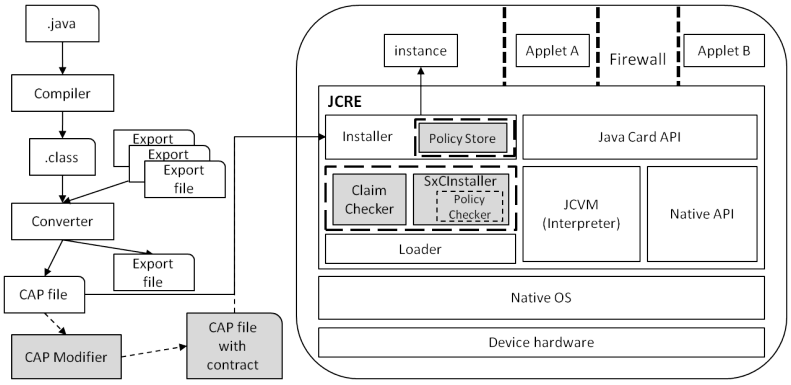


Fig. 1. *The Java Card architecture and the loading process. The white components and data structures belong to the standard Java Card platform. The grey components and data structures are additions introduced by the* S×C *scheme. The dashed lines denote the changes to the loading process.*

of specific *shareable interfaces*, also called *services*. A shareable interface is an interface that extends `javacard.framework.Shareable`.

An application A implementing some services is called a *server*. An application B that tries to call any of these services is called a *client*. A typical scenario of service usage starts with a client's request to the JCRE for a reference to A's object (that is implementing the necessary shareable interface). The firewall passes this request to application A, which decides if the reference can be granted or not. If the decision is positive, the reference is passed through the firewall and is stored by the client for further usage. The client can now invoke any method declared in the shareable interface which is implemented by the referenced object. During invocation of a service a *context switch* will occur, thus allowing invocation of a method of the application A from a method of the application B. A call to any other method, not belonging to a shareable interface, will be stopped by the Java Card firewall.

As all applet interactions inside one package are not controlled by the firewall and due to the fact that a package is loaded in one pass, we consider that one package contains only one applet and there is an one-to-one correspondence between packages and applications. Another important assumption for us is that packages do not implement shareable interfaces declared in other packages, this assumption can in fact be guaranteed by the S×C framework.

Currently the services access control enforcement on Java Card is embedded into the application code. Traditionally, the server will receive an AID (unique application identifier) of the client requesting its service from the JCRE and check that this client is authorized before granting it the reference to the object (that can implement multiple services). Once the object reference is received, the client can access all the services within this object and it can also leak the

object reference to other parties. The S×C framework checks the authorizations for each service access, thus the object reference leaks are no longer a security threat. In Java Card the controls for checking the invocation context can also be embedded directly in the code of each service. We argue that this approach is not satisfactory, as the access control list of authorized clients can be updated only through complete removal and reinstallation of the server applet. When the server package is imported by other (client) packages on the card the server removal is not possible. The embedded S×C verifier can enforce the same service access control policies in a flexible fashion when each server can update its policy without reinstallation.

In a CAP file all object types and methods are referred to by their tokens which are used by the JCRE for on-card linking. A service s is identified as a tuple $\langle A, I, t\rangle$, where A is the AID of the package providing the service s, I is a token for a shareable interface where the service is defined and t is a token for the method in the interface I. The correct service tokens can be obtained from the Export file of a package (produced by the Converter) or from the CAP file.

The JCRE imposes some restrictions on method invocations in the application bytecode. Only the opcode `invokeinterface` allows to perform the context switch between two different packages. Thus, in order to collect all potential service invocations we analyze the bytecode and infer from the `invokeinterface` instructions possible services to be called. More details are available for the interested reader in [5].

3 Security-by-Contract for Java Cards

In the Security-by-Contract scheme every application carries its contract embedded into the CAP file. Let $A.s$ be a service s declared in a package A. The contract consists of two parts: AppClaim and AppPolicy. AppClaim specifies provided and invoked services (Provides and Calls sets correspondingly). We say that the service $A.s$ is provided if applet A is loaded and service s exists in its code. Service $B.m$ is invoked by A if A may try to invoke $B.m$ during its execution. The AppClaim will be verified for compliance with the bytecode (the CAP file).

The application policy AppPolicy contains authorizations for services access (sec.rules set) and functionally necessary services (func.rules set). We say a service is necessary if a client will not be functional without this service on board. The AppPolicy lists applet's requirements for the smart card platform and other applications loaded on it. A functionally necessary service for applet A is the one which absence on the platform will crash A or make it useless. For example, a transport application normally requires some payment functionality to be available. If a customer will not be able to purchase the tickets, she would prefer not to install the ticketing application from the very beginning. It is required that for every application A $\mathsf{func.rules_A} \subseteq \mathsf{Calls_A}$. An authorization for a service access contains the package AID of the authorized client and the service tokens. The access rules have to be specified separately for each service and each client that the server wants to grant access.

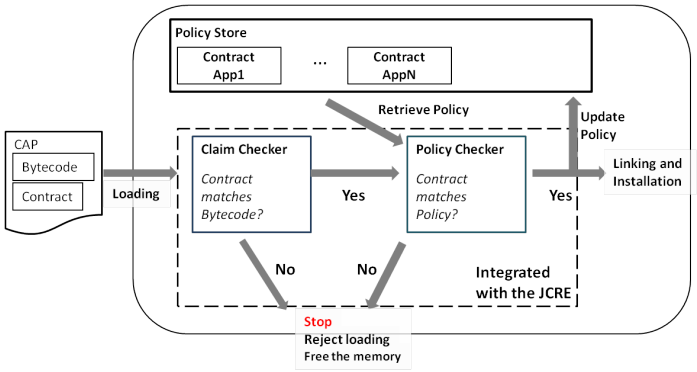


Fig. 2. *The Security-by-Contract workflow for loading.*

Contracts are delivered on the card within Custom components of the CAP files. CAP files carrying Custom components can be recognized by any Java Card Installer, as the Java Card specs require. More details on the structure of the Contract Custom component that we proposed are available in [5]. To ease the contract creation we have developed the CAP Modifier tool with a user-friendly graphical interface allowing to edit any section of the contract, save already created contracts as files for future usage and embed the created contracts into CAP files. The tool takes the CAP file generated with the standard Java Card tools and appends the Contract Custom component within it, modifying the Directory component of the CAP file accordingly (as the specification requires).

The S×C framework deployed on the card consists of two main components integrated with the platform: the ClaimChecker and the PolicyChecker. The ClaimChecker performs extraction of the contract and verifies that it is compliant with the application code. Then the PolicyChecker ensures that the security policy of the card, composed by all the contracts of currently loaded applications, is compliant with the contract. Another addition to the platform is the PolicyStore component. The PolicyStore appears due to the fact that only components implemented in Java Card (applets and the Installer) can allocate space in EEPROM (mutable persistent memory), that is the only type of memory suitable to store the security policy across the card sessions. The PolicyStore is a class in the Installer. Figure 1 depicts the S×C prototype embedded into the Java Card platform. Figure 2 summarizes the S×C workflow for loading, as the most interesting case, emphasizing the role of each component.

4 Implementation of the S×C Prototype

We have implemented the S×C prototype in C, as it is a standard language for smart card platform components implementation and the Loader API we had knowledge of was written in C. The main components of the S×C prototype are:

SxCInstaller. This component is an interface with the Installer. SxCInstaller calls the ClaimChecker that in a positive case (contract and bytecode are compliant) will return the address of the contract in the Contract Custom Component of the CAP file being loaded. The SxCInstaller also comprises (for memory saving reasons) the PolicyChecker component.

ClaimChecker. This component is called by SxCInstaller. It carries out the check for the compliance between the contract and the CAP file. The check is carried out after parsing the CAP file. We used a part of the standard Loader API, called in the current paper CAPlibrary, that contains functions to access the beginning and the length of each CAP file component. Using the functions of the CAPlibrary library for CAP file parsing on-card, this component gets the initial address of the components it needs from which it can eventually parse the rest of the components. If the result is positive, the ClaimChecker will return the address of the contract of the application in the Contract Custom component.

The ClaimChecker component has to establish that the services from $\mathsf{Provides}_\mathsf{A}$ exist in package A and the services from $\mathsf{Calls}_\mathsf{A}$ are indeed the only services that A can try to invoke in its bytecode. The goal of the ClaimChecker algorithm is to collect each `invokeinterface` opcode with its parameters (the method t and the Constant Pool index I). Then the collected set is compared with the set Calls of the contract. We emphasize that operands of the`invokeinterface` opcode are known at the time of conversion into a CAP file and thus are available directly in the bytecode. All methods of the application are provided in the Method Component of the application's CAP file, an entry for each method contains an array of its bytecodes. Exported shareable interfaces are listed in the Export component of the CAP file and flagged in the Class component. The strategy for the ClaimChecker is to ensure that each service listed in the Provides set is meaningful and no other provided services exist. More details of the ClaimChecker algorithm can be found in [5].

Due to the lack of space we only present the security policy data structures just to give a flavor of this part of the system. The security policy stored on the card consists of the contracts of the currently loaded applications. A contract in the form supplied on the card is a space-consuming structure (each AID can occupy up to 16 bytes). Therefore we have resolved to store the security policy on the card in a bit vectors format. The current data structure for security policy assumes there can be up to 10 loaded applets, each containing up to 8 provided services. Thus the security policy is a known data structure called Policy with a fixed format, the bits are taking 0 or 1 depending if the applet is loaded or the service is called/provided. The Mapping object maintains correspondence between the number the applet gets in the on-card security policy structure and the actual AID of the package, and between the provided service token and the number of this service in the policy data structure. The other two objects that are part of the on-card security policy are the MayCall list and WishList list, containing the potential future authorizations, necessary for a case when a loaded application carries a security rule for some application not yet on the card and the services that are called by applications but are not yet on the

card, because the server is not yet loaded, or because the current version of the server does not provide this service correspondingly. The PolicyStore, being written in Java Card, has to communicate the security policy to the PolicyChecker component (the SxCInstaller) that will run the contract-policy compliance check. This communication is implemented through usage of a native API.

The Developer S×C Prototype. The S×C prototype for experimenting on a PC comprises the same components: the SxCInstaller, the ClaimChecker and the PolicyStore, which in the Developer version is packaged as an applet. The communication between the SxCInstaller and the PolicyStore applet is emulated by using files. The S×CDeveloper prototype emulates deployment of the PolicyStore applet on a card using the Java Card development kit from Oracle. For the purposes of independent functionality testing we have implemented the CAPlibrary library and the necessary Installer functionality following the JCRE specifications. The Developer prototype accepts as input CAP files with the contract embedded into the Custom component by the CAP Modifier tool, runs the verification algorithms and outputs the results, notifying also which of the checks failed during verification (if any, otherwise it reports successful loading of an applet and updates the current security policy). Thus developers can create and embed different contracts and try the verification process.

5 Evaluation

In this section we report the memory measurements of the prototype carried out in the University of Trento. The details of the industrial evaluation performed by Trusted Labs (commissioned by Gemalto) can be found in [1]. The most important characteristics for an on-card component are RAM and non-volatile memory (NVM) consumption. NVM space is required to store the prototype and the necessary data (the security policy) across the card sessions. RAM memory is used to store the temporary data while the verification is performed.

We have explored two metrics for the NVM usage estimations off-device: the size of the object files in C compiled on a PC and the number of lines of code (LOCs). The ClaimChecker component object file compiled with the MinGW compiler tools occupies 6.5 KB, the ClaimChecker has 170 LOCs (.h + .c). The SxCInstaller object file occupies 7.3 KB, this component includes 178 LOCs. The PolicyStore applet CAP file (exact on-device measure) occupies 6KB.

RAM usage is also very important, as over-consumption of RAM by the prototype can lead to the denial of service. The higher is the RAM consumption, the lower is the level of interoperability of the prototype; because some cards cannot provide a significant amount of RAM for the verifier which has to run in the same time with the Installer. We have used a temporary array of 255 bytes to store the necessary computation data. 255 bytes is a small temporary memory buffer which ensures the highest level of interoperability for the prototype.

6 Conclusions

In the paper we have presented the S×C prototype for the Java Card-based smart cards. The prototype can be used to perform load time verification of Java Card applets and enforce service access control policies in a flexible way. The prototype is integrated on the card with the Loader API which provides direct access to the received bytecode. The prototype is able while parsing the bytecode to extract the application contract and to compare it with the actual code of the application. Then the received contract is transformed into the memory-efficient on-card format and compared with the security policy of the device. The memory statistics demonstrates feasibility of embedding the proposed prototype on an actual device. We have also developed the Developer S×C prototype that can be demonstrated on a usual PC without actual device, all the on-card functions necessary to process CAP files were implemented using the Java Card specifications.

Acknowledgements. We thank Quang-Huy Nguyen and Boutheina Chetali for insights on Java Card. This work was partially supported by the EU under grants EU-FP7-FET-IP-SecureChange and FP7-IST-NoE-NESSOS.

References

1. M. Angeli, G. Bergmann, P. Capelastegui, B. Chetali, O. Delande, M. Felici, F. Innerhofer-Oberperfler, V. Meduri, F. Paci, S. Paul, B. Solhaug, and A. Tedeschi. D1.3: Report on the industrial validation of SecureChange solutions. *SecureChange EU project public deliverable, www.securechange.eu*, 2012.
2. P. Bieber, J. Cazin, V. Wiels, G. Zanon, P. Girard, and J-L. Lanet. Checking secure interactions of smart card applets: Extended version. *J. of Comp. Sec.*, 10(4):369–398, 2002.
3. N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Security-by-Contract: towards a semantics for digital signatures on mobile code. In *Proc. of EuroPKI-07*, volume 4582 of *LNCS*, pages 297 – 312. Springer-Verlag, 2007.
4. A. Fontaine, S. Hym, and I. Simplot-Ryl. On-device control flow verification for java programs. In *ESSOS'2011*, volume 6542 of *LNCS*, pages 43–57. Springer.
5. O. Gadyatskaya, E. Lostal, and F. Massacci. Load time security verification. In *ICISS'2011*, volume 7093 of *LNCS*, pages 250–264. Springer.
6. D. Ghindici and I. Simplot-Ryl. On practical information flow policies for java-enabled multiapplication smart cards. In *Proceedings of CARDIS 2008*, volume 5189 of *LNCS*, pages 32–47. Springer-Verlag, 2008.
7. M. Huisman, D. Gurov, C. Sprenger, and G. Chugunov. Checking absence of illicit applet interactions: a case study. In *FASE'04*, volume 2984 of *LNCS*, pages 84–98. Springer-Verlag, 2004.
8. Sun Microsystems. Virtual Machine and Runtime Environment. Java Card^{TM} platform. Specification 2.2.2, Sun Microsystems, 2006.
9. G. Schellhorn, W. Reif, A. Schairer, P. Karger, V. Austel, and D. Toll. Verification of a formal security model for multiapplicative smart cards. In *ESORICS'00*, volume 1895 of *LNCS*. Springer-Verlag, 2000.

Study, Formalisation, and Analysis of Dalvik Bytecode

Henrik Søndberg Karlsen, Erik Ramsgaard Wognsen, Mads Chr. Olesen, and René Rydhof Hansen

Department of Computer Science, Aalborg University
{hkarls07,ewogns08}@student.aau.dk
{mchro,rrh}@cs.aau.dk

Abstract. With the large, and rapidly increasing, number of smartphones based on the Android platform, combined with the open nature of the platform that allows "apps" to be downloaded and executed on the smartphone, misbehaving and malicious (malware) apps is set to become a serious problem. To counter this problem, automated tools for analysing and verifying apps are essential. Further, to ensure high-fidelity of such tools, we believe that it is essential to formally specify both semantics and analyses.
In this paper we present the first (to the best of our knowledge) formalisation of the Dalvik bytecode language and formally specified control flow analysis for the language. To determine which features to include in the formalisation and analysis, 1,700 Android apps from the Android Market were downloaded and examined.

1 Introduction

With over 100 million Android devices already activated, and in excess of 400,000 new activations every day, Android has become one of the most widespread and fastest growing computing platforms for smartphones and tablet computers. The combination of the wide distribution and the open nature of the Android platform, where *apps* can be downloaded and installed not only from the official Android Market but also from unknown, untrusted, and potentially malicious third parties makes it obvious that tools are needed to ensure, and possibly certify, that apps are well-behaved and do not access (and leak) information or functionality not explicitly allowed and intended by the user. The problem is further exacerbated by the often sensitive and private nature of information stored on a smartphone as well as the potential for apps to (ab-)use services that cost the user money, e.g., by secretly sending text messages to expensive premium numbers [11].

In order to develop tools for highly trustworthy analysis and, especially, for certification we believe it is necessary to have a formal underpinning of the target platform and to show that the analyses are sound with respect to the formalisation. In this paper we first present a study of 1,700 Android apps, carried out in order to determine what Dalvik instructions and language features are

most often used in typical apps. Based on the results of this study, we develop a formal operational semantics for the Dalvik bytecode language [14]. We further abstract the operational semantics into a formal *control flow analysis* for the Dalvik bytecode language, intended both as the basis for further, more specialised analyses but also by itself for detecting potentially malicious actions, e.g., leaking prviate information or surreptitiously calling expensive phone numbers. To the best of our knowledge, this is the first such formalisation of the Dalvik bytecode language and a accompanying control flow analysis. Finally, since our study revealed that more than half the apps examined used reflection, we illustrate how the *reflection* API can be formalised and analysed.

While Android apps are generally developed in Java, compiled to Java bytecode, and only then converted to Dalvik bytecode, we focus here on Dalvik bytecode because it is the common executable format for all Android apps and, therefore, offers the best opportunity for performing analyses as close to the code actually executed as possible and we sidestep issues relating to decompiling and reverse engineering apps, cf. [4].

1.1 Related Work

In [3] the tool *ComDroid* is described as a tool that performs "flow sensitive, intraprocedural static analysis with limited interprocedural analysis" of Dalvik bytecode programs. It is designed to analyse the communication between Android applications through the so-called *Intents*, the Android equivalent of events, and to find potential security vulnerabilities in the communication patterns of applications. In [6] the ComDroid tool is used as a component of another analysis tool, called *Stowaway*, that analyses API calls in applications to determine if they are over-privileged. In order to improve the precision and efficacy of the analysis, Stowaway incorporates some analysis of the reflection features found in Dalvik bytecode (through the `java.lang.reflect` library). Both ComDroid and Stowaway are sophisticated analysis tools covering not only the Dalvik bytecode language but also important parts of the API and the Android platform itself. However, since the analyses are not actually specified in detail, neither formally nor informally, it is impossible to evaluate the exact strengths and weaknesses of the underlying analyses. Indeed, it is stated in [6] that Stowaway makes a "primarily linear traversal" and that it "experiences problems with non-linear control flow". This emphasises the need for a formalisation of both the Dalvik bytecode language as well as the control flow analysis.

In [4] Android applications are analysed by first recovering the Java source through decompilation and then using the Fortify SCA static analysis tool to detect potential security vulnerabilities. While the paper reports on impressive results using this approach, it is also noted that it was not possible to recover the source code for all the targeted applications and thus making analysis of those applications impossible. Analysing directly at the bytecode sidesteps this issue.

The approach described in [13] takes advantage of the fact that most, if not all, Android applications are developed in Java and adapts the Julia framework

for Java bytecode analysis to the specificities of the Java bytecode that results from developing Android applications (before being converted to Dalvik bytecode). The Julia framework is theoretically sound and well-documented, but the described solution requires access to the Java bytecode version of an application in order to analyse it.

2 Study of Apps

In order to identify which Dalvik bytecode instructions and which Java language features are used in typical Android apps, we have collected and examined 1,700 of the most popular free apps from Android Market [7]. Notable features include code obfuscation, threading, reflection, native libraries, and dynamic class loading. The apps were collected in November 2011 using Android 2.3.3 on a Samsung Nexus S.

For efficiency reasons the Dalvik bytecode language contains several specialised variants of many common instructions, e.g., there are numerous variants of the `move` instruction. For our study we have grouped instruction variants that are semantically similar, e.g., most variants of the `move` instruction belong to the same group. In the semantics (see Section 3) we use the same notion of grouping to abstract and generalise the original 218 Dalvik bytecode instructions into a set of 39 instructions.

In our study we found that, with the exception of the `filled-new-array` instruction, *all* types of Dalvik bytecode instructions are used in more than half the studied apps. In particular, the instructions `invoke-direct` and `return-void` are used in every app and even the most rare instructions, `sparse-switch` and `filled-new-array`, are used in 69.7% and 22.3% of the studied apps, respectively. The instructions that occur most frequently are `invoke-virtual` and `move-result`, which are used more than 12 million times each in total in the 1,700 apps. In comparison, `filled-new-array` is used 1,930 times. For full details, see [10].

The observations made from studying the use of Java features are summarised in Table 1 and are explained in detail below. For the study we have separated code into *developer code* and *library code*. Developer code is code that lies within the natural packages for the application. For an application `company.app` this means all classes located directly in the packages `/`, `/company/`, `/company/app/`, and any subpackages in `/company/app/`. Library code is everything else.

Code obfuscation, especially using ProGuard [5], is used to a large extent. We searched for classes named "`a`" within apps in the data set, and used this as an approximation to determine if an app contains any obfuscated code. The same approach was used in [4] which found 36% of apps to include obfuscated code. We found the class in 64.82% of the apps. Obfuscation has legitimate uses and is recommended by Google [8], but makes it harder to manually inspect the code.

Table 1. Percentages of apps in our data set that use various features.

Feature	Used by apps	Hereof in libraries
Obfuscated source	64.82%	-
Has native libraries	20.35%	-
`java/lang/Thread`	90.18%	24.07%
`java/lang/reflect`	73.00%	55.92%
`java/lang/ClassLoader`	39.71%	81.19%
`java/lang/Runtime;->exec`	19.53%	80.44%

Native libraries, i.e., ARM shared object (`.so`) files, were included in 20.35% of the apps we studied[1]. A previous study [4] found that of their 1,100 studied apps from September 2010, only 6.45% included shared objects. We presume the increased usage is because the Android NDK, released June 25, 2009, has gained more widespread use in 2011.

Threading, as indicated by the use of monitors, i.e., the Java `synchronized` keyword, was found in 88% of the apps. Furthermore, 90.18% of the apps include a reference to `java/lang/Thread`. These observations are not conclusive, but indicate that multi-threaded programming is wide-spread. However, further studies are needed to substantiate the results.

Reflection is used extensively in Android apps for accessing private and hidden classes, methods, and fields, for JSON and XML parsing, and for backward compatibility [6]. We confirmed these observations by manual inspections. Of the 940 apps studied in [6], 61% were found to use reflection, and using automated static analysis they were able to resolve the targets for 59% of the reflective calls. 73% of the apps in our data set use reflection. This indicates that a formalisation of reflection in Dalvik is necessary to precisely analyse most apps. We treat this in Section 5.

Class Loading Of the studied apps 39.71% contain a reference to the class loader library, `java/lang/ClassLoader`, which means that the app can load Dalvik executable (DEX) files and JAR files at runtime. Manual inspection shows that some of these uses relate to IPC transport with the Android Parcelable interface.

The Java method `Runtime.exec()` is used to execute programs in a separate native process and is present in 19.53% of the apps. We manually inspected some of these uses. Most of these do not use a hardcoded string as the argument to `exec`, but of those that do, we found execution of both the `su` and `logcat` programs which, if successful, give the app access to run programs as the super user on the platform or read logs (with private data [4]) from all applications, respectively.

[1] In addition, 15 apps included the ARM executable `gdbserver`.

3 Operational Semantics

In this section we describe the formalisation of the Dalvik bytecode language using operational semantics. With the exception of instructions related to concurrency, we have formalised the full (generalised) instruction set and below we present the semantic rules for a few interesting instructions. The approach is inspired by a similar effort to formalising the Java Card bytecode language [15, 9].

To simplify our work, we have made a number of convenient, but minor, generalisations, including: simplified type hierarchy to avoid dealing with bit-level operations, except when absolutely necessary; "inlining" of the constant pools for easier and more direct reference of strings, types, methods, and fields; and finally an idealised program counter abstracting away the length of instructions. While none of these modifications change the expressive power of a Dalvik application, they greatly simplify presentation and formalisation.

The study described in Section 2 impacted the formalisation in two major ways: it was clear that all of the core bytecode language had to be formalised and also that the reflection API had to be formalised. In order to ensure that the formalisation correctly represents the Dalvik (informal) semantics, we based the formalisation on the documentation for Dalvik [1], inspection of the source code for the Dalvik VM in Android [2], tests of handwritten bytecode, and experiments with disassembly of compiled Java code.

3.1 Program Structure and Semantic Domains

To facilitate the development of the formal semantics for Dalvik bytecode, it is important to have a good formalisation of the program structure of apps. Here we follow [15] and use domains equipped with accessor-functions, written in an OO inspired notation. The Method domain is presented to illustrate this approach:

$$\begin{aligned}\mathsf{Method} = {} & (\mathit{name}\colon \mathsf{MethodName}) \times (\mathit{class}\colon \mathsf{Class}) \times \\ & (\mathit{argType}\colon \mathsf{Type}^*) \times (\mathit{returnType}\colon \mathsf{Type} \cup \{\texttt{void}\}) \times \\ & (\mathit{instructionAt}\colon \mathsf{PC} \to \mathsf{Instruction}) \times \\ & (\mathit{kind}\colon \{\texttt{virtual}, \texttt{static}, \texttt{direct}\}) \times \\ & (\mathit{maxLocal}\colon \mathbb{N}_0) \times (\mathit{handlers}\colon \mathbb{N}_0 \to \mathsf{ExcHandler})\end{aligned}$$

The return type of a method $m \in \mathsf{Method}$ is denoted $m.\mathit{returnType}$. The function *argType* is a sequence of the types of the arguments to the method. The function *instructionAt* maps to another function mapping locations in the method (program counter values) to instructions. *maxLocal* is the number of the last register used for local variables in the method (Dalvik uses registers instead of an operand stack).

The next step in our formalisation is to define the semantic domains. Since these are quite standard, and for lack of space, we only give a few examples. Local registers are modelled simply as a map from register names to values using $\perp$ to

denote undefined register contents: $\mathsf{LocalReg} = (\mathbb{N}_0 \cup \{\texttt{retval}\}) \to \mathsf{Val}_\perp$. Note that a special register, the `retval` register, is used to transfer return values from invoked methods.

Addresses are used to identify specific program points and are composed of a method and a program counter value: $\mathsf{Addr} = \mathsf{Method} \times \mathsf{PC}$ where $\mathsf{PC} = \mathbb{N}_0$. This gives a unique address for every instruction in a Dalvik program. We can then define stack frames to contain a method and a program counter, i.e., an address, and the local registers: $\mathsf{Frame} = \mathsf{Method} \times \mathsf{PC} \times \mathsf{LocalReg}$. This leads to the following definition of call stacks as simply a sequence of frames except that the top frame may be an *exception frame* representing an as yet unhandled exception: $\mathsf{CallStack} = (\mathsf{Frame}+\mathsf{ExcFrame}) \times \mathsf{Frame}^*$. An exception frame contains the location of its corresponding exception object on the heap and the address of the instruction that threw the exception: $\mathsf{ExcFrame} = \mathsf{Location} \times \mathsf{Method} \times \mathsf{PC}$. We shall not go into further details with the semantic domains, merely refer to [10] for the full definition.

3.2 Semantic Rules

We specify the semantics as a straightforward structural operational semantics where each configuration comprises a static heap, a heap, and a call stack (as defined above). To illustrate the semantics, we present the semantic rule for the central `invoke-virtual` instruction:

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invoke-virtual}\ \ v_1 \ldots v_n\ \ meth \\ R(v_1) = loc \qquad loc \neq \texttt{null} \qquad o = H(loc) \\ n = arity(meth) \qquad m' = resolveMethod(meth, o.class) \\ R' = [0 \mapsto \perp, \ldots, m'.maxLocal \mapsto \perp, \\ m'.maxLocal + 1 \mapsto v_1, \ldots, m'.maxLocal + arity(m') \mapsto v_n] \end{array}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m', 0, R'\rangle :: \langle m, pc, R\rangle :: SF\rangle}$$

The function *instructionAt* is an access function on the Method domain that identifies the instruction at a given location in a specified method. In the new configuration, the static heap (S) and dynamic heap (H) are unchanged. The instruction receives n arguments and the signature of the method to invoke. The first argument v_1 is a reference to the object on which the method should be invoked. The location of the method is resolved using the auxiliary function *resolveMethod* as explained below and this method is put into a new frame on top of the call stack, with the program counter set to 0. A new set of local registers, R', is created, where the registers up to $m'.maxLocal$ are mapped to $\perp_{\mathsf{Val}}$ such that they are initially undefined. The arguments are then mapped into the next registers. Like Java, Dalvik implements dynamic dispatch. We define a function to search through the class hierarchy for virtual methods (all non-static

methods that are overridable or defined in interfaces):

$$resolveMethod(meth, cl) = \begin{cases} \bot & \text{if } cl = \bot \\ m & \text{if } m \in cl.methods \wedge meth \lhd m \wedge \\ & \quad m.kind = \texttt{virtual} \\ resolveMethod(meth, cl.super) & \text{otherwise} \end{cases}$$

where $meth \lhd m$ is a predicate formalising when a method signature $meth$ is compatible with a given method $m \in \mathsf{Method}$, i.e. when the names, argument types and return types match.

Exceptions can be thrown either explicitly using the `throw` instruction, or by the system in case of a runtime error, such as a null dereference. Exception handlers have a type for the exceptions it may catch, a program counter value pointing to the handler code, and program counter values defining the boundaries of the region covered by the exception handler. Exceptions that are not handled in the method where it is thrown are put on the call stack for the next method's exception handlers to try to handle. The `throw` instruction is defined as follows:

$$\frac{m.instructionAt(pc) = \texttt{throw}\ v \qquad R(v) = loc_E \neq \texttt{null} \qquad cl = H(loc_E).class \qquad cl \preceq \texttt{Throwable}}{A \vdash \langle S, H, \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle loc_E, m, pc\rangle :: \langle m, pc, R\rangle :: SF\rangle}$$

When the top frame is an exception frame and a handler is found, the following rule applies (an analogous rule applies when no handler is found):

$$\frac{cl = H(loc_E).class \qquad findHandler(m, pc, cl) = pc'}{A \vdash \langle S, H, \langle loc_E, m_E, pc_E\rangle :: \langle m, pc, R\rangle :: SF\rangle \Longrightarrow \langle S, H, \langle m, pc', R[\texttt{retval} \mapsto loc_E]\rangle :: SF\rangle}$$

The auxiliary function *findHandler* finds the exception handler matching the location in the method and the given exception class or $\bot$ if no appropriate handler is available.

4 Control Flow Analysis

In the following we give a very brief overview of the control flow analysis, which is specified using flow logic [12]. In this approach an analysis is defined through a number of flow logic judgements that specify what is required of an analysis result in order to be correct. Details can be found in [12]. Many other frameworks for program analysis exist, but the combination of a structural operational semantics and flow logic has proven to be flexible and easy to use for both theoretical developments as well as for implementation. A more detailed comparison with other approaches is out of scope for this paper.

The abstract domains, used in the analysis to statically represent runtime values, are based very closely on the underlying semantic domains. Similar to the *class object graphs* in [16], we map all instances of a given class to one

abstract class instance. In order to achieve sufficient precision, the analysis is flow-sensitive, but only intra-procedurally. This yields an abstract domain for local registers that tracks (abstract) values for every address in the program (including a special address, denoted END, used to track return values from methods): $\widehat{\mathsf{LocalReg}} = \overline{\mathsf{Addr}} \to (\mathsf{Register} \cup \{\mathsf{END}\}) \to \widehat{\mathsf{Val}}$ where $\widehat{\mathsf{Val}}$ is the domain for abstract values, containing abstractions of primitive types, references and classes. Similarly we can define the abstract domains for static and "ordinary" heaps: $\widehat{\mathsf{StaticHeap}} = \mathsf{Field} \to \widehat{\mathsf{Val}}$ and $\widehat{\mathsf{Heap}} = \overline{\mathsf{Ref}} \to (\widehat{\mathsf{Object}} + \widehat{\mathsf{Array}})$ respectively. The overall domain for the analysis is then defined as $\widehat{\mathsf{Analysis}} = \widehat{\mathsf{StaticHeap}} \times \widehat{\mathsf{Heap}} \times \widehat{\mathsf{LocalReg}}$.

We now illustrate flow logic judgements, starting with the judgement for the `move` instruction: after a `move` instruction, the destination register contains the value in the source register while all others are unchanged as signified by the $\sqsubseteq_{\{v\}}$ relation:

$$\begin{aligned}(\hat{S}, \hat{H}, \hat{R}) \models (m, pc)\colon &\texttt{move}\ \ v_1\ \ v_2 \\ \text{iff}\quad &\hat{R}(m, pc)(v_2) \sqsubseteq \hat{R}(m, pc+1)(v_1) \\ &\hat{R}(m, pc) \sqsubseteq_{\{v_1\}} \hat{R}(m, pc+1)\end{aligned}$$

The conditions are joined by an implicit conjunction. A solution for the analysis that satisfies the ordering specified by the flow logic judgements will be a safe over-approximation of all possible values in every method. To satisfy the conditions for the `move` instruction, the value of the destination register at the following instruction must be the least upper bound of the old and the new value.

The flow logic judgement for the `invoke-virtual` instruction works as follows: For each possible object the method can be called on, the method is resolved (by dynamic dispatch using the *resolveMethod* function from the semantics), the arguments are transferred, and the `retval` register is updated with the return value unless the return type of the method is void:

$$\begin{aligned}(\hat{S}, \hat{H}, \hat{R}) \models (m, pc)\colon &\texttt{invoke-virtual}\ \ v_1 \ldots v_n\ \ meth \\ \text{iff}\quad &\forall(\mathsf{ObjRef}\ \ cl) \in \hat{R}(m, pc)(v_1)\colon \\ &\quad m' = resolveMethod(meth, cl) \\ &\quad \forall 1 \le i \le n\colon \hat{R}(m, pc)(v_i) \sqsubseteq \hat{R}(m', 0)(m'.maxLocal + i) \\ &\quad m'.returnType \neq \texttt{void} \Rightarrow \hat{R}(m', \mathsf{END}) \sqsubseteq \hat{R}(m, pc+1)(\texttt{retval}) \\ &\hat{R}(m, pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m, pc+1)\end{aligned}$$

Object references are modelled simply as classes, which is possible since these are all known statically.

Two things can happen when an exception is thrown: If a local handler exists, control is transferred to that handler with a reference to the exception object in the `retval` register. If no local handler exists, the method aborts and the exception is put on the call stack in an exception frame. The analysis will treat this situation with the following auxiliary predicate:

$$\begin{aligned}&\mathsf{HANDLE}_{(\hat{R}, \hat{E})}((\mathsf{ExcRef}\ \ cl_E), (m, pc\ )) \equiv \\ &\quad findHandler(m, pc, cl_E) = pc' \Rightarrow \{(\mathsf{ExcRef}\ \ cl_E)\} \sqsubseteq \hat{R}(m, pc')(\texttt{retval}) \\ &\qquad\qquad\qquad\qquad\qquad\quad \hat{R}(m, pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m, pc') \\ &\quad findHandler(m, pc, cl_E) = \bot \Rightarrow \{(\mathsf{ExcRef}\ \ cl_E)\} \sqsubseteq \hat{E}(m)\end{aligned}$$

where the exception cache $\hat{E}$ abstracts the details of exceptions on the call stack by storing them when no local handler is found. With the above predicate it is now trivial to define the analysis for the `throw` instruction:

$$
\begin{array}{l}
(\hat{S},\hat{H},\hat{R},\hat{E}) \models (m,pc)\colon \texttt{throw}\ \ v \\
\quad \text{iff} \quad \forall(\mathsf{ExcRef}\ \ cl_E) \in \hat{R}(m,pc)(v)\colon \\
\qquad\quad \mathsf{HANDLE}_{(\hat{R},\hat{E})}((\mathsf{ExcRef}\ \ cl_E),(m,pc))
\end{array}
$$

5 Reflection

As shown by our study, many apps use reflection and it is therefore important to handle this in the formalisation and analysis. Below we show how a central API method in the reflection library is formalised and analysed, namely method invocation. Refer to [10] for further details.

Dynamic method invocation is done by calling the `invoke()` method on a `java.lang.reflect.Method` object. This method is used by 84.3% of the apps that use reflection. Modelling the semantics of this results in the following special case for the `invoke-virtual` instruction:

$$
\frac{\begin{array}{c}
m.\mathit{instructionAt}(pc) = \texttt{invoke-virtual}\ \ v_1\ \ v_2\ \ v_3\ \ \mathit{meth} \\
\mathit{meth}.\mathit{name} = \texttt{Method.invoke} \qquad R(v_1) = \mathit{loc}_1 \qquad \mathit{loc}_1 \neq \texttt{null} \\
o_1 = H(\mathit{loc}_1) \qquad o_1.\mathit{class} \preceq \texttt{Method} \qquad \mathit{meth}' = \mathit{methodSignature}(o_1) \\
R(v_2) = \mathit{loc}_2 \qquad \mathit{loc}_2 \neq \texttt{null} \qquad o_2 = H(\mathit{loc}_2) \qquad R(v_3) = \mathit{loc}_3 \qquad a = H(\mathit{loc}_3) \in \mathsf{Array} \\
m' = \mathit{reflectResolveMethod}(\mathit{meth}', o_2.\mathit{class}) \qquad R' = [0 \mapsto \bot, \ldots, m'.\mathit{maxLocal} \mapsto \bot, \\
m'.\mathit{maxLocal}+1 \mapsto a.\mathit{value}(0), \ldots, m'.\mathit{maxLocal} + a.\mathit{length} \mapsto a.\mathit{value}(a.\mathit{length}-1)]
\end{array}}{A \vdash \langle S,H,\langle m,pc,R\rangle :: SF\rangle \Longrightarrow \langle S,H,\langle m',0,R'\rangle :: \langle m,pc,R\rangle :: SF\rangle}
$$

Register v_1 points to the `Method` object, v_2 points to the receiver object, i.e., the object on which the method is invoked, and v_3 points to an array with the arguments for the method. The rule uses two auxiliary functions: *methodSignature* extracts the semantic method signature from a `Method` object and *reflectResolveMethod* works like *resolveMethod* except that it ignores the method kind such that it works on any method.

The flow logic judgement is similar to the one for `invoke-virtual`, with another layer of indirection because the over-approximative analysis represents sets of possible values in registers:

$$
\begin{array}{l}
(\hat{S},\hat{H},\hat{R}) \models (m,pc)\colon \texttt{invoke-virtual}\ \ v_1\ \ v_2\ \ v_3\ \ \mathit{meth} \\
\quad \text{iff} \quad \mathit{meth}.\mathit{name} = \texttt{Method.invoke} \\
\qquad \forall(\mathsf{ObjRef}\ \ cl) \in \{cl \mid cl \in \hat{R}(m,pc)(v_1) \wedge cl \preceq \texttt{Method}\}\colon \\
\qquad\quad \forall \mathit{meth}' \in \mathit{methodSignature}(\hat{H}(cl))\colon \\
\qquad\qquad \forall(\mathsf{ObjRef}\ \ cl') \in \hat{R}(m,pc)(v_2)\colon \\
\qquad\qquad\quad m' = \mathit{reflectResolveMethod}(\mathit{meth}', cl') \\
\qquad\qquad\quad \forall 1 \leq i \leq \mathit{arity}(\mathit{meth}')\colon \\
\qquad\qquad\qquad \forall(\mathsf{ArrRef}\ \ a) \in \hat{R}(m,pc)(v_3)\colon \\
\qquad\qquad\qquad\quad \hat{H}(\mathsf{ArrRef}\ \ a) \sqsubseteq \hat{R}(m',0)(m'.\mathit{maxLocal}+i) \\
\qquad\qquad\quad m'.\mathit{returnType} \neq \texttt{void} \Rightarrow \hat{R}(m',\mathsf{END}) \sqsubseteq \hat{R}(m,pc+1)(\texttt{retval}) \\
\qquad \hat{R}(m,pc) \sqsubseteq_{\{\texttt{retval}\}} \hat{R}(m,pc+1)
\end{array}
$$

In effect, the conditions of the basic `invoke-virtual` must be satisfied for each method that can be resolved from each `Method` object reference in v_1. Furthermore, the contents of each array referenced by v_3 must be available in each argument register for the current method being invoked.

This approach presents a problem: For the *methodSignature* function to work, the `Method` object must be known statically and no classes must be loaded or created dynamically. In the studied apps, the `Method` object is typically derived from a `Class` object using the `getMethod()` method which takes the method name as a string argument. The `Class` object itself is also typically obtained from a string using the `Class.forName()` method. In many cases we have determined that the string can be found statically. Of the apps that use `Method.invoke()`, 18.9% use only local string constants for `forName()` and `getMethod()` or get the `Class` and `Method` objects from the `const-class` Dalvik instruction or simple API methods such as the `getClass()` instance method.

We discovered that many of the apps that we could not classify as using static strings only used strings of unknown origin in a single case: A Google AdMob library for install referrer tracking. This library calls a number of `BroadcastReceiver`s and as a safe over-approximation, we expect to include this as a special case in the analysis by letting it call all available broadcast receivers. With this addition, we would be able to analyse the use of reflection in 36.7% of the apps.

The above numbers are based on primitive intra-procedural data tracking and with the implementation of the analysis and its inter-procedural (but flow insensitive) data flow, they should be improved.

6 Conclusion

In this paper we have shown excerpts of a formal semantics for the Dalvik bytecode language and a formally specified control flow analysis, both based on the results of a study of 1,700 Android apps. Also based on this study, we have identified reflection as a particularly important language feature (supported through the `java.lang.reflect` API) to take into account when formalising semantics and analysis.

In future work we will finish the formal proof of soundness for the analysis (work in progress, currently a few cases have been proved) and also implement a prototype of the control flow analysis, by systematically converting the flow logic judgements into Prolog terms.

References

1. Android Open Source Project: Bytecode for the Dalvik VM. http://source.android.com/tech/dalvik/dalvik-bytecode.html (December 13th 2011)
2. Android Open Source Project: Downloading the Source Tree. http://source.android.com/source/downloading.html (December 14th 2011)

3. Chin, E., Felt, A., Greenwood, K., Wagner, D.: Analyzing inter-application communication in android. In: Proc. of the Annual International Conference on Mobile Systems, Applications, and Services (2011)
4. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A study of Android application security. In: In proc. of the 20th USENIX Security Symposium (SEC'11). pp. 315–330. USENIX Association, San Francisco, CA, USA (Aug 2011)
5. Eric Lafortune: ProGuard. http://proguard.sourceforge.net (December 13th 2011)
6. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: Proceedings of the 18th ACM conference on Computer and communications security. pp. 627–638. CCS '11, ACM, New York, NY, USA (2011)
7. Google: Android Market. https://market.android.com (November 29th 2011)
8. Google Inc.: ProGuard | Android Developers. Web page available at http://developer.android.com/guide/developing/tools/proguard.html (December 13th 2011)
9. Hansen, R.R.: Flow Logic for Language-Based Safety and Security. Ph.D. thesis, Technical University of Denmark (2005)
10. Karlsen, H.S., Wognsen, E.R.: Study, Formalisation, and Analysis of Dalvik Bytecode. Master's thesis, Aalborg University (2012), forthcoming. Preliminary version available at http://students.cs.aau.dk/erikrw/report_sw9.pdf
11. Alienvault Labs: Analysis of Trojan-SMS.AndroidOS.FakePlayer.a. Web page available at http://labs.alienvault.com/labs/index.php/2010/analysis-of-trojan-sms-androidos-fakeplayer-a/ (November 29th 2011)
12. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer Verlag (1999)
13. Payet, É., Spoto, F.: Static analysis of Android programs. In: Proc. of the 23rd International Conference on Automated Deduction (CADE-23). Lecture Notes in Computer Science, vol. 6803, pp. 439–445. Springer Verlag, Wroclaw, Poland (Jul/Aug 2011)
14. The Android Open Source Project: Bytecode for the Dalvik VM. Retrieved 2011-10-13 from http://source.android.com/tech/dalvik/dalvik-bytecode.html
15. Siveroni, I.: Operational Semantics of the Java Card Virtual Machine. Journal of Logic and Algebraic Programming 58(1–2), 3–25 (Jan/Mar 2004)
16. Vitek, J., Horspool, R.N., Uhl, J.S.: Compile-Time Analysis of Object-Oriented Programs. In: Proc. International Conference on Compiler Construction (CC'92). Lecture Notes in Computer Science, vol. 641. Springer Verlag (1992)