Project N°: **FP6-015905**

Project Acronym: **MOBIUS**

Project Title: **Mobility, Ubiquity and Security**

Instrument: **Integrated Project**

Priority 2: **Information Society Technologies**

Future and Emerging Technologies

Deliverable D4.7

Report on on-device checking

Due date of deliverable: 2009-08-31 (T0+48)

Actual submission date: 2009-10-15

Start date of the project: **1 September 2005** Duration: **48 months**

Organisation name of lead contractor for this deliverable: **INRIA**

Revision 8092

Project co-funded by the European Commission in the Sixth Framework Programme (2002-2006)		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Contributions

Site	Contributed to Chapter
FT	2
INRIA	1,3
TL	4
UPM	5

This document was written by Frédéric Besson (INRIA), Pierre Crégut (FT), Thomas Jensen (INRIA), Mariela Pavlova (TL), and Germán Puebla (UPM).

Executive Summary

This document summarises Deliverable 4.7 on On-device Proof Checking of project FP6-015905 (MOBIUS), co-funded by the European Commission within the Sixth Framework Programme. Full information on this project, including the contents of this deliverable, is available online at `http://mobius.inria.fr/`.

Contents

Chapter 1

Introduction: On-device checking

Proof-carrying code (PCC) is based on the idea that in a security-critical code transmission setting, the code producer should provide some evidence that the program she distributes is safe and/or functionally correct. The code consumer would thus receive the program together with a certificate (proof) that attests that the program has the desired properties.

The objective of the current Task 4.5 is to investigate methods and techniques for facilitating the checking of program certificates on resource-constrained devices; in particular mobile telephones. The approach has been to use the so-called Abstraction-Carrying Code approach in which the result of static, automatic data flow analysis is used to produce certificates of program properties. The properties under consideration have included points-to and alias information, data dependency and flow of string values, as well as resource-oriented analyses aimed at computing the complexity of a program with respect to a resource (time, space, number of network accesses, sending of SMSs *etc*).

Focus has been on obtaining certificate checkers that are efficient, reliable and can be run on-device. Increased reliability has partly been achieved by extracting checkers form certified Coq specifications. The extracted checkers must be run by the end-user before he uses the application to check that the security policy is enforced. When the application is a midlet, this could be done off-line on a regular desktop, but it is preferable that the verification is a quick, mandatory phase performed directly on the mobile handset before the first use of the application, in a similar way as byte code verification. To keep checking reasonably efficient and certificate size small, techniques for compression of fixpoints issued by data flow analysis have been developed.

Chapter 2 describes a points-to analysis for Java byte code derived from the Datalog analysis framework by Whaley and Lam. This analysis both provides useful information about data flow and alias, and provides a rudimentary string analysis.

Chapter 3 describes techniques for compressing invariants obtained from polyhedral analysis of Java byte code, by pruning of fixpoints and by removing redundant information that the checker will re-compute anyway. The analysis can among other things be used to certify certain, linear resource usages of applications.

Chapter 4 describes how the analysis tool TL SAT developed by Trusted Logic can be used to derive program certificates with information about flow of strings and dependencies between program variables. The latter information provides a simple information flow analysis.

Chapter 5 presents two developments which are needed in order to perform practical on-device checking of resource bounds. The first one allows obtaining asymptotic upper bounds from (non-asymptotic) cost relations. The second one allows checking user-provided resource usage bounds with those inferred by static analysers such as COSTA.

Chapter 2

Two phase points-to analysis

2.1 An on-demand points-to analysis based on a context sensitive analysis expressed in Datalog

2.1.1 Whole program analysis

A points-to analysis [4, 9] computes an over-approximation of the storage locations that can be pointed to by pointer variables at a given execution point. Storage locations are represented by the points where they are allocated and eventually the execution context of this allocation. Execution points are usually approximated by the program point and eventually some approximation of the call stack.

Datalog is a fragment of Prolog without function symbols (there are no complex terms) and negation. Programs on finite domains can be efficiently compiled as operations on binary decision diagrams coding the relations.

John Whaley [11, 12] has formalized context sensitive analysis as a Datalog program where initial relations describe the initial assignment of variables with newly created objects (represented by allocation sites), how values flows between method parameters, local variables and object fields and the initial knowledge on how method calls are resolved. The program computes new relations describing how the allocated objects are propagated to other variables. Contexts (call stacks) are directly coded as atoms. A relation describes how the call stack may evolve at each invocation point depending on the resolution of the virtual call. Call stacks that only differ because of recursive calls are represented by a single atom.

2.1.2 On-demand analysis

The on-demand analysis was first developed as an extraction algorithm that prunes the fixpoint computed by the whole analysis (represented by the computed relations). Two kinds of relations are computed:

- the fact that a variable or a field of an abstract object contributes to the points-to value of another variable because its contents can be transferred to this variables (eventually through a sequence of assignments),

- the fact that an allocation site can flow to a variable or an object field.

Those relations are computed only for the variables and allocation sites necessary to compute the points-to set of the variables we are interested in.

In fact this algorithm computes on the fly the points-to set of the variables used as the base object for accessing fields we are interested in and we do not need any of the results of the global analysis. So the algorithm can be used as an independent on-demand analysis. Details are presented in an annex [**?**].

2.2 A checker for the on-demand points-to analysis

The certificate is coded as a tabular representation of the relations computed by the on-demand analysis and by the class hierarchy analysis used to solve virtual method calls.

The checker linearly scan the byte code of the application and checks locally on each instruction that the constraints expressed in the certificate are fullfilled:

- that the knowledge a variable points-to set is needed for another variable points-to set is correctly propagated backward with respect to the data flow,
- that an object from a given allocation site can be stored in a given variable is propagated forward with respect to the data flow,
- that the resolution of virtual method is coherent with the propagation between arguments at method invocation points and parameters of actual methods.

2.3 Extension to string analysis

Points-to analysis is a key component for verifying security related properties of applications. The most obvious example is string analysis that can be used to check constraints on strings used as parameters of security relevant methods.

In MIDP, the main usage is the definition of URLs that describe the protocol and the address of network connections. This has a direct impact on their cost and can also raise other security issues if the phone connects to a rogue site.

A string analysis can be easily developed from a points-to analysis because strings are not mutable [7]. String concatenation is performed through StringBuffer objects that are usually local to a method. The scheduling of operations on StringBuffers inside a method can be retrieved from a simple intra-procedural analysis. The propagation of string objects thorough the program is computed by the points-to analysis.

2.4 Implementation choices

Certified checkers are developed as Coq terms and their actual source code is extracted from the Gallina source as Objective Caml programs. The complete checker for the points-to analysis was directly developed in Objective Caml following the pattern of those certified checkers so that a certified version can be eventually be derived.

Those Objective Caml programs are cross-compiled to native code for the ARM processor with a modified compiler based on Eric Cooper cross compiler for Objective Caml with float support disabled (because it is incompatible with current hardware and not required). Compiling the Objective Caml runtime on a Linux based platform is easier than on non POSIX compliant environment (Symbian, Windows for mobile).

Due to all those constraints, targetting a Linux based phones was an obvious choice. We used a Motorola A780with the open source "EZX crosstool" compilation chain.

Because we do not have a direct access to the native UI libraries of the phone, the user interface of the application is implemented as a midlet. It is quite slow : launching a midlet takes several seconds and is usually slower than our analyzers. Benchmarks have been performed with a command line version of the analyzers accessed through a telnet connection with the phone.

2.5 Benchmarks

The prototype targets calls to `Connector.open`. We have used a set of 75 small free midlets from the midlets.org site [8] that open network connections to evaluate the behaviour of our analysis.

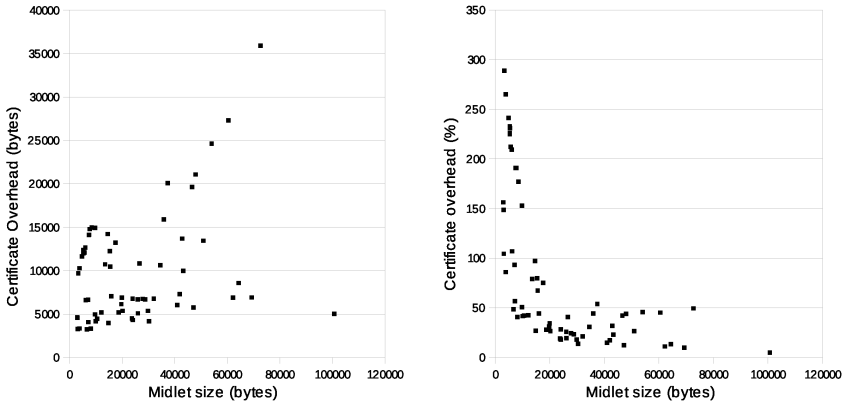

Figure 2.1: Size of certificates

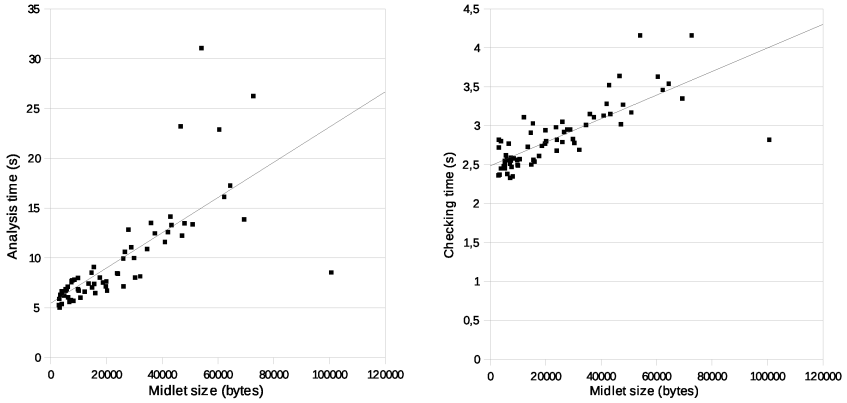

Figure 2.2: Performance of the analysis on PC and of checking on the handset

Figure 2.1 shows the size of certificates. They are often in the 5k-15k range but some of them are bigger. The overhead of the certificate is high for small midlets but is still reasonable (usually 20%, some around 50% in rare cases)for bigger ones. It is clearly still too high for an industrial deployment. We have not yet tried to improve its size. Potential solutions are:

- using a binary format and using a single file to improve zip compression,
- transfering some information directly in the class to reduce the size of the certificate constant pool (around 30% of the certificate) that duplicates the class constant pool.

Figure 2.2 gives performance result for the on-demand analysis on a PC (1.7 GHz Pentium-M based laptop) and for checking it on a mobile (A780). The checking time on the handset is usually between 2 and 3s (around 0.4 s when run on the laptop). It could be reduced if the checker did not verify MIDP classes that are not used by the midlet. The performance of this non optimised code is already good enough for an industrial deployment.

Analysis time is more correlated to the size of midlets than checking time because the bulk of the analysis is in the code of the application. This is not true for the checking phase that is almost proportional to the size of the code base whatever is the computational content of the analysed code.

Chapter 3

Small certificates for on-device proof-carrying code

The work of INRIA in this work-package has focused on the problem of how to produce small certificates for proof-carrying code. This problem has been studied in the setting of abstraction-carrying code[3] where the program certificates are obtained from the outcome of a static analysis. Such analyses will produce program invariants in the form of fixpoints of abstract transfer functions and the present work has been concerned with several aspects of how to simplify and reduce the size of these fixpoints:

- Abstract interpretations will often produce more information that necessary for proving a particular program property. We have defined the notion of a **witness** of a property together with a technique for **pruning** program invariants to provide the minimum information necessary to prove a particular property.
- The pruning algorithm has been specialised for the particular case of polyhedral-based abstract interpretation. Using results from linear programming we have shown how to efficiently compute the smallest subset of linear constraints of a polyhedron needed to prove a given property. This allows a greedy pruning algorithm for which each pruning step is locally optimal.
- The checking of a proposed invariant involves a certain amount of re-computing the invariant. It is therefore possible to reduce the amount of information that is being transmitted in the certificate because the checker will reconstruct it anyway. We have developed general **fixpoint reconstruction** algorithms that generalize the dedicated algorithms from lightweight bytecode verification.
- An essential part of verifying a proposed invariant is the checking of polyhedral inclusions. We have developed a notion of certificate for checking polyhedral inclusions based on a use of Farkas' lemma that reduces inclusion checking to a few simple matrix computations. This work has been documented in the deliverable D4.4 of Task 4.3 of this work-package, entitled "Generation of certificates". The certificates described in D4.4 are complete for $\mathbb{R}$ with full addition and multiplication but incomplete for linear arithmetic over $\mathbb{Z}$. We have proposed and implemented in Coq an extended certificate format that is also complete for linear arithmetic over $\mathbb{Z}$.

Fixpoint pruning was first presented in [6] in the case of polyhedral invariants and for Java stack maps with interfaces in [5]. The PhD thesis of Tiphaine Turpin [10] provides a comprehensive description of the technique and its applications.

3.1 Witnesses and invariant pruning

An abstract interpretation of a program can be described by an abstract domain D whose elements represent sets of potential execution states of the program, and an abstract transfer function $F : D \to D$ that over-approximates the possible steps that can be taken. A program invariant is a post-fixpoint of F.

A *safety property* can be given by an element ϕ of D, which represents a set of states that are safe to enter. If w satisfies $w \sqsubseteq \phi$, then this means that all reachable states are safe, and therefore the program is safe. Furthermore, w is itself a *witness* whose existence is sufficient to ensure the program's safety, and thus can play the role of certificate for PCC. Checking that a given abstract element w is indeed a witness only amounts to *checking* the two above constraints, namely, $F(w) \sqsubseteq w$ and $w \sqsubseteq \phi$, which is fundamentally simpler that *computing* w.

Definition 3.1.1 *A witness for a property $\phi \in D$ and a (monotone) abstract operator $F : D \to D$ is an abstract property $w \in D$ such that*

$$F(w) \sqsubseteq w \sqcap \phi.$$

In the case of distributive data flow analyses, it can be shown that an optimal witness can be copmputed. However, the more complex (and therefore useful) program analyses are generally not distributive. For the general case, we assume that abstract states are described as sets of constraints, as is the case in polyhedral analysis. The intuitive idea of pruning is then to remove as many constraints as possible from a given witness (which is computed by standard fixpoint iteration).

3.2 Efficent pruning for polyhedral-based abstract interpretation

Given polyhedra P and Q – viewed as sets of linear inequations – and a linear transfer function F such that $F(P) \sqsubseteq Q$, the problem consists in finding a minimal subset of inequations of P, say P', such that $F(P') \sqsubseteq Q$. The properties of the pruning are that P' is smaller than P but strong enough to ensure the property Q. The problem can be written as

$$min\{P' | P' \subseteq P, F(P') \sqsubseteq Q\}$$

Because F is a linear transformation, this minimisation problem can be solved by minimising a set of emptiness problems of the form

$$min\{P' | P' \subseteq P, P \cup D_i \sqsubseteq \emptyset\}$$

where D_i is obtained from F and the ith inequation of Q.

Using Farkas's Lemma, we have shown that P' can be obtained from the vertices of dual polyhedron of $P \cup D_i$.

This result allows for an efficient and locally optimal greedy pruning algorithm for polyhedron-based abstract interpretation.

3.3 Fixpoint reconstruction

In the context of PCC based on abstract interpretation, a part of a witness can usually be deduced, or *reconstructed* from the rest using the abstract transfer function. We have developed a reconstruction algorithm which takes advantage of our definition of witness, and a method for generating corresponding certificates. For a program given by a control flow graph, and given a witness, we propose to derive a certificate of the form (K, S) where K gives the value of the witness for a subset of the program points, and S specifies an "iteration strategy", that is, a sequence of program points whose corresponding local abstract transfer functions are to be evaluated for reconstructing the witness.

Definition 3.3.1 *A certificate is a pair (K, S) where $K : [1, n] \mapsto D$ is a* partial *mapping from program points to properties and $S \in [1, n]^*$ is a sequence of program points.*

The interpretation of such a certificate is given by the following reconstruction algorithm. The soundness of this algorithm is established by the following Theorem:

Theorem 3.3.2 *Let (K, S) be a certificate. If check(K, S) succeeds then the program satisfies the associated security property ϕ.*

check(K, S) =
  check that every j defined in K appears at least once in S
  let $w \in D^n$ be defined as $w = \left[j \mapsto \begin{cases} K(j) \text{ if } K(j) \text{ is defined} \\ \top \text{ otherwise} \end{cases} \right]$
  for j iterating over S do
    compute $w'_j = F_j(\Pi_j(w))$
    check that $w'_j \sqsubseteq w_j$
    $w_j \leftarrow w'_j$
  done
  check that $w \sqsubseteq \phi$

3.4 Enhanced certificate format for linear artihmetic

Program verification often requires arithmetic reasoning over $\mathbb{Z}$. We have designed compact and easily checkable certificates asserting that a set of polynomial inequations do not have an integer solution. The basic block of the certificate is a certificate for real arithmetic asserting that a set of polynomial inequations do not have real solutions. Indeed, if there is no real solution, there is no integer solution. The purpose of the enhanced certificate is to assertthe absence of integer solution even if there are real solutions (*e.g.*, $2 \times x = 1$). In order to handle proofs over $\mathbb{Z}$, the certificate format provides two additional constructions allowing to obtain *cutting planes* and *case splits*. A cutting plane is an inequation that can be strengthened by rounding the constant over the nearest integer.

Theorem 3.4.1 (Cutting Plane) *Let p be an integer and c a rational constant.*

$$p \geq c \Rightarrow p \geq \lceil c \rceil$$

A case split allows to enumerate over all the possible integer values over a finite interval.

Theorem 3.4.2 (Enumeration) *Let p be an integer and c_1 and c_2 integer constants.*

$$c_1 \leq p \leq c_2 \Rightarrow \bigvee_{x \in [c_1, c_2]} p = x$$

In pseudo-ML syntax[1], the enhanced certificate format `zcert` is the following.

```
type zcert =
 | RealProof of rcert * zcert?
 | CutProof of rcert * zcert?
 | EnumProof of rcert * rcert * (zcert list)
```

where `rcert` is the type of certificate for real arithmetic.

The checker takes as argument a list of hypotheses. It evaluates the certificate to construct logic consequences of the initial inequalities that are added to the current hypotheses. The checker stops as soon as it detects a logic consequence that is trivially contradictory *e.g.*, $-1 \geq 0$.

- If the certificate has the form `RealProof(rcert,zcert)`, the checker computes using `rcert` a logic consequence of the hypotheses and continues with `zcert`.
- If the certificate has the form `CutProof(rcert, zcert`, the checker computes using `rcert` a logic consequence $p \geq c$. It computes the cutting plane $p \geq \lceil c \rceil$ and continues with `zcert`.

[1] In the type definition, the question mark denotes an optional argument

- If the certificate has the form `EnumProof(rcert1,recer2,zcerts)`, the checker computes a logic consequence $c_1 \leq p$ using `rcert1` and $p \leq c_2$ using `rcert2`. Using Theorem 3.4.2, it starts a case split with the list of certificate `zcerts`. The list has length $c_2 - c_1 + 1$. The ith element of the list is a proof the the hypotheses augmented with $p = c_1 + i - 1$ is a contradiction.

This enhanced certificate format is complete for linear integer arithmetic and is part of Coq 8.2.

Chapter 4

PCC infrastructure with the static analysis tool TL SAT

TL SAT is a static analysis tool tailored to bytecode programs and is mainly used for the certification of mobile or embedded applications prior to their installation on the user devices. More particularly the tool has been used for the security validation of Java Card and Midlet applications. The properties verified by the tool concern the use of critical APIs which vary from forbidding the use of some APIs to constraining the shape of their parameters as well as simple information flow properties. The need for such validation is explained with the fact that the Midp or JavaCard framework (and in general Java) do not provide mechanisms to check such kind of properties. The validation performed by TL SAT however is not a guarantee that the code shipped on the user device really respects the properties verified by TL SAT. Of course, frameworks like Midp rely on digital signatures which guarantee that the code is not tempered and identify the issuer of the application. Although the combination of security certification and digital signatures seems to close safely the certification chain, currently there is no way to verify on the user device prior to applet execution that the latter is complying to the device security policy.

To fill in this gap, we study the possibility for a PCC certification chain based on TL SAT. As any PCC infrastrtucture, the one that we propose relies on certificates accompanying the untrusted code. The certificate generation is done outside the device by TL SAT. Certificates bring an evidence that the code adheres to a certain number of rules and are checked on the device.

An important factor for the feasability of a PCC infrastruture depends on the format and the size of the certificate accompanying the untrusted code. Therefore, the main purpose of this study is the investigation of these certificate characteristics. The proposed certificate consists of an overapproximation of the runtime application behavior of the certified program. The overapproximation is the result of the of TL SAT which is based on abstract interpretation algorithms. Compared to certificates based on deductive program verification, this approach has the benefit to generate certificates automatically without the need of user interaction (e.g. writing annotations or proving verification conditions). Note that automation is an important parameter in an industrial context as the constraints on the development and certification phase are very strong. Of course, TL SAT is not as expressive as specification languages like JML and is not as powerful as logical verification techniques but provides a sufficient coverage for many interesting security properties.

4.1 Certificate format

Designing the format of a PCC certificate is an important step as the certificate will greatly impact the feasibility and scaling up of the whole PCC infrastructure. The certificate format must be efficiently checkable, i.e. it should contain sufficient information so that the checking procedure on the client side be sufficiently fast. The size of the certificate is also important as it has a direct impact on the certificate transportation over the network and storage on the device. On the other hand, small certificates imply that they do not

contain much information and therefore their corresponding checker is potentially more complex. In what follows we discuss several issues which are important for designing a PCC infrastructure which scales up.

We propose that the certificate in a PCC infrastructure based on TL SAT contains overapproximations for the argument Arg^{cert} and return values Ret^{cert} of each live method, a global overapproximation $Heap^{cert}$ of the heap used by the application and finally overapproximation of states related to special program points of the method execution. The first three components allow for modular verification of the application code - every method `m` will be inspected by assuming the precondition $Arg^{cert}(\texttt{m})$ and by checking the postcondition $Ret^{cert}(\texttt{m})$; for every method call to `m`, the precondition $Arg^{cert}(\texttt{m})$ is assumed and the postcondition $Ret^{cert}(\texttt{m})$ is assumed. The fourth component allows for efficient verification of a method implementation where each instruction in a method bytecode is visited at most once by the certificate checker.

applet	jar size (kb)	certificate of method pre and post state, heap (kb)	compressed certificate (kb)
CellHtml	48	524	48
WebViewer	48	528	48
MobileMule	56	516	52
Kungfu	76	420	38
MGMaps	96	692	64
J2MeMap	196	872	76
MyWiowa	216	896	72
AplusBegalX	268	1274	100
Itransports	352	568	49

Table 4.1: Statistics on the certificate and application sizes

We performed measurements over the results returned by TL SAT which are shown in Table 4.1. The experiment consisted in running the analysis over several applications and serialising the produced approximations for the method argument and return values as well as the heap. We have chosen a set of applications with various functionalities and sizes in order to obtain a representative average for the certificate size w.r.t. the complexity of the application. The first column in the table shows the name of each of the applications that we analysed, the second column shows the size of the application jar file, the third column gives the size of the certificate, i.e. the serialised information. The last column shows the size of the compressed certificate. The indicated application size is the size of the applet jar file as it is shipped on the device. The size of the serialized analysis result (third column from left to right) is quite large w.r.t. the size of the application size- the average certificate size is around 4 times larger than the applet size. However, the compressed version of the certificate is in average 2 to 3 times smaller than the applet size which implies that the compressed certificate increases in average the code package with up to 30% which is a positive result. It is also interesting to note that while for small applets (up to 60Kb) the compressed certificate has almost the same size as the applet size, for larger applets the size of the compressed certificate is often 2 to 3 times smaller than the applet size. For the largest applet (352 Kb) the certificate compression is 7 times smaller (49 Kb). Such drastic reductions in the certificate size can be explained also with the fact that the applet contains additional resources like images or sound files. In this experiment we do not take into account the part of the certificate concerning backedges in the code. We expect that the size of this information would have a small impact on the overall certificate size.

4.1.1 Certificate checker

The certificate checker is composed from an abstract Java virtual machine enriched with checks against the device security policies and checks over the correctness of the certificate accompanying the application code. When one of the checks fails the whole certificate check fails. The checker has a linear complexity w.r.t. the size of the certified code and is consequently efficient. We do not enter here in details about the algorithm underlying the checker but it does not defer from from existing work in the field. For more details the reader may check the full version of the TL contribution for the Mobius final technology evaluation.

4.2 Conclusion

We have investigated the possibility for building a PCC infrastructure for safe resource use and simple information policies based on the static analysis tool TL SAT. Concerning the certificate format, our experiments are positive as they show that the size of a certificate containing sufficient information to perform one pass on-device checking is times smaller than the code size itself. This is an important point as PCC infrastructures relying on heavy certificate formats are in practice not acceptable in a realistic scenario. Another point which is crucial for building a PCC architecture is the on device checker the implementation of which is the next step before providing a full fledged PCC infrastructure.

Chapter 5

Checking Resource Usage Bounds

An important application of resource analysis is *resource certification*, whereby programs are coupled with information about their resource usage. This information allows deciding whether the resources used by the program execution are acceptable or not *before* running the program. In this sense, resource usage can be considered a security property of untrusted mobile code, possibly in the context of proof-carrying code. Programs whose resource usage is not certified are potentially harmful, since their execution may require more resources than we are willing to spend or they may even have monetary cost by executing *billable events* such as sending text messages or making http connections on a mobile phone. In fact, mobile devices is one of the settings where resource certification is more important, because of the limited computing power typically available on mobile devices.

Ideally, one would like to extend bytecode verification in order to include more sophisticated *security policies* in the static verification part. This problem can be formulated in two ways. One is to have an automatic system which given a program and a resource usage policy answers *yes* only if it succeeds to prove that the program satisfies the policy. Alternatively, we can split this process in two steps: first, an automatic system obtains an upper bound on the resource usage of the program and second, another automatic system, which in what follows we refer to as *comparator*, checks whether the computed upper bound is smaller than or equal to the resource usage policy for any possible input value. We advocate for the second alternative because we believe it is more flexible: we first use COSTA on the code producer side to infer upper bounds which are independent of any resource policy and consumer, and then, on the code consumer side we check whether the upper bound abides by the policy.

We illustrate through a simple example the fundamental intuition behind resource certification. Let us assume a resource usage policy for method $m(t)$ that imposes a resource usage policy, which we call *policy*, on the number of instructions executed of:

$$policy{=}60 * [\mathsf{nat}(t)]^2 + 120 * \mathsf{nat}(t) + 13$$

COSTA infers the upper bound $ub{=}24 * \mathsf{nat}(t) * \lceil \mathsf{log}_2(\mathsf{nat}(t) + 1) \rceil + 53 * \mathsf{nat}(t) + 12$. The code will be acceptable, provided that *policy* is guaranteed, i.e., $ub \leq policy$, which happens to be the case in our example and that the comparator succeeds to prove it.

We have developed a comparator which handles closed-forms that involve logarithmic, exponential, polynomial expressions, etc. It is described in Section 5.2 below.

Also, though in some contexts, especially when considering memory usage, non-asymptotic policies are to be expected, sometimes it is more reasonable that the policy is asymptotic. In order to allow the use of asymptotic policies, COSTA has been extended in order to compute asymptotic upper bounds. This is described in Section 5.1 below. Coming back to our previous example, this would result in a new $policy'$ s.t. $policy'{=}[\mathsf{nat}(t)]^2$. The comparator should again be able to prove that $policy'$ is satisfied by method m.

5.1 Asymptotic Resource Usage Bounds

A fundamental characteristics of a program is the amount of resources that its execution will require, i.e., its *resource usage*. Typical examples of resources include execution time, memory watermark, amount of data transmitted over the net, etc. *Resource usage analysis* aims at automatically estimating the resource usage of programs. Static resource analyzers often produce *cost bound functions*, which have as input the size of the input arguments and return bounds on the resource usage (or *cost*) of running the program on such input.

A well-known mechanism for keeping the size of cost functions manageable and, thus, facilitate human manipulation and comparison of cost functions is *asymptotic analysis*, whereby we focus on the behaviour of functions for large input data and make a rough approximation by considering as equivalent functions which grow at the same rate w.r.t. the size of the input date. The asymptotic point of view is basic in computer science, where the question is typically how to describe the resource implication of scaling-up the size of a computational problem, beyond the "toy" level. For instance, the big O notation is used to define *asymptotic upper bounds*, i.e, given two functions f and g which map natural numbers to real numbers, one writes $f \in O(g)$ to express the fact that there is a natural constant $m \geq 1$ and a real constant $c > 0$ s.t. for any $n \geq m$ we have that $\mid f(n) \mid \leq c* \mid g(n) \mid$. Other types of (asymptotic) computational complexity estimates are lower bounds ("Big Omega" notation) and asymptotically tight estimates, when the asymptotic upper and lower bounds coincide (written using "Big Theta"). The aim of *asymptotic resource usage analysis* is to obtain a cost function f_a which is *syntactically simple* s.t. $f_n \in O(f_a)$ (correctness) and ideally also that $f_a \in \Theta(f_n)$ (accuracy), where f_n is the non-asymptotic cost function. Besides, as we will develop in the paper, f_a can be computed by obtaining f_n first and then simplifying it into f_a or, more interestingly, be produced directly.

The scopes of non-asymptotic and asymptotic analysis are complementary. Non-asymptotic bounds are required for the estimation of precise execution time (like in WCET) or to predict accurate memory requirements. The motivations for inferring asymptotic bounds are twofold: (1) They are essential during program development, when the programmer tries to reason about the efficiency of a program, especially when comparing alternative implementations for a given functionality. (2) Non-asymptotic bounds can become unmanageably large expressions, imposing huge memory requirements. We will show that asymptotic bounds are syntactically much simpler, can be produced at a smaller cost, and, interestingly, in cases where their non-asymptotic forms cannot be computed.

The main techniques presented in [1] are applicable to obtain asymptotic versions of the cost functions produced by any cost analysis, including lower, upper and average cost analyses. Besides, we will also study how to perform a tighter integration with an upper bound solver which follows the classical approach to static cost analysis. In this approach, the analysis is parametric w.r.t. a *cost model*, which is just a description of the resources whose usage we should measure, e.g., time, memory, calls to a specific function, etc. and analysis consists of two phases. (1) First, given a program and a cost model, the analysis produces *cost relations* (CRs for short), i.e., a system of recursive equations which capture the resource usage of the program for the given cost model in terms of the sizes of its input data. (2) In a second step, *closed-form*, i.e., non-recursive, upper bounds are inferred for the CRs.

How the first phase is performed is heavily determined by the programming language under study and nowadays there exist analyses for a relatively wide range of languages and their references). Importantly, such first phase remains the same for both asymptotic and non-asymptotic analyses and thus we will not describe it. The second phase is language-independent, i.e., once the CRs are produced, the same techniques can be used to transform them to closed-form upper bounds, regardless of the programming language used in the first phase. The important point is that this second phase can be modified in order to produce asymptotic upper bounds directly. Our main contributions in [1] can be summarized as follows:

1. We adapt the notion of *asymptotic complexity* to cover the analysis of realistic programs whose limiting behaviour is determined by the limiting behaviour of its loops. The latter often depends on linear combinations of several program arguments which might increase or decrease.

2. We present a novel transformation from *non-asymptotic cost functions* into asymptotic form. After some syntactic simplifications, our transformation detects and eliminates subterms which are *asymptotically subsumed* by others and preserves the complexity order. When using our transformation as a back-end of any non-asymptotic cost analyzer for average cost, upper or lower bounds, we accomplish motivation (1) above.

3. In order to achieve motivation (2), we need to integrate the above transformation within the process of obtaining the cost functions. We present a tight integration into (the second phase of) a resource usage analyzer to generate directly asymptotic upper bounds without having to first compute their non-asymptotic counterparts.

4. We report on a prototype implementation within the COSTA system which shows that we are able to achieve motivations (1) and (2) in practice.

5.2 Comparing Cost Functions in Resource Analysis

In all applications of resource analysis, such as resource-usage verification, program synthesis and optimization, etc., it is necessary to compare cost functions. This allows choosing an implementation with smaller cost or to guarantee that the given resource-usage bounds are preserved.

Essentially, given a method m, a cost function f_m and a set of linear constraints ϕ_m which impose size restrictions (e.g., that a variable in m is larger than a certain value or that the size of an array is non zero, etc.), we aim at comparing it with another cost function bound $\mathtt{b}$ and corresponding size constraints $\phi_{\mathtt{b}}$. Depending on the application, such functions can be automatically inferred by a resource analyzer (e.g., if we want to choose between two implementations), one of them can be user-defined (e.g., in resource usage verification one tries to verify, i.e., prove or disprove, *assertions* written by the user about the efficiency of the program).

From a mathematical perspective, the problem of cost function comparison is analogous to the problem of proving that the difference of both functions is a decreasing or increasing function, e.g., $\mathtt{b} - f_m \geq 0$ in the context $\phi_{\mathtt{b}} \wedge \phi_m$. This is undecidable and also non-trivial, as cost functions involve non-linear subexpressions (e.g., exponential, polynomial and logarithmic subexpressions) and they can contain multiple variables possibly related by means of constraints in $\phi_{\mathtt{b}}$ and ϕ_m. In order to develop a practical approach to the comparison of cost functions, we take advantage of the form that cost functions originating from the analysis of programs have and of the fact that they evaluate to non-negative values. Essentially, our technique, described in [2], consists in the following steps:

1. Normalizing cost functions to a form which make them amenable to be syntactically compared, e.g., this step includes transforming them to sums of products of basic cost expressions.

2. Defining a series of comparison rules for basic cost expressions and their (approximated) differences, which then allow us to compare two products.

3. Providing sufficient conditions for comparing two sums of products by relying on the product comparison, and enhancing it with a *composite* comparison schema which establishes when a product is larger than a sum of products.

We have implemented our technique in the COSTA system. Our experimental results demonstrate that our approach works well in practice, it can deal with cost functions obtained from realistic programs and verifies user-provided upper bounds efficiently.

Bibliography

[1] E. Albert, D. Alonso, P. Arenas, S. Genaim, and G. Puebla. Asymptotic resource usage bounds. In *Programming Languages and Systems: Proceedings of the 7th Asian Symposium APLAS 2009*, Lecture Notes in Computer Science. Springer-Verlag, 2009. To appear.

[2] E. Albert, P. Arenas, S. Genaim, I. Herraiz, and G. Puebla. Comparing cost functions in resource analysis. In *FOPARA '09: Proceedings of the International Workshop on Foundational and Practical Aspects of Resource Analysis*, 2009.

[3] E. Albert, P. Arenas, and G. Puebla. Incremental certificates and checkers for abstraction-carrying code. In *Workshop on the Issues in the Theory of Security*, March 2006.

[4] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, May 1994.

[5] Frédéric Besson, Thomas Jensen, and Tiphaine Turpin. Computing stack maps with interfaces. In *Proc. of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, volume 5142 of *Lecture Notes in Computer Science*, pages 642–666. Springer-Verlag, 2008.

[6] Frédéric Besson, Thomas P. Jensen, and Tiphaine Turpin. Small witnesses for abstract interpretation-based proofs. In *Programming Languages and Systems: Proceedings of the 16th European Symposium on Programming, ESOP 2007*, number 4421 in Lecture Notes in Computer Science, pages 268–283. Springer-Verlag, 2007.

[7] P. Crégut and C. Alvarado. Improving the security of downloadable Java applications with static analysis. In *Bytecode Semantics, Verification, Analysis and Transformation*, volume 141 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2005.

[8] MOBIUS Consortium. Deliverable 5.1: Selection of case studies, 2007. Available online from `http://mobius.inria.fr`.

[9] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Principles of Programming Languages*, pages 32–41, New York, NY, USA, 1996. ACM Press.

[10] Tiphaine Turpin. *Pruning program invariants*. PhD thesis, Univ. Rennes 1, 2008.

[11] John Whaley. *Context-Sensitive Pointer Analysis using Binary Decision Diagrams*. PhD thesis, Stanford University, March 2007.

[12] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Programming Languages Design and Implementation*. ACM Press, June 2004.