The MOBIUS Proof Carrying Code Infrastructure
(An Overview)

Gilles Barthe\(^1\), Pierre Crégut\(^3\), Benjamin Grégoire\(^2\), Thomas Jensen\(^4\),
and David Pichardie\(^4\)

\(^1\) IMDEA Software, Madrid, Spain
\(^2\) INRIA Sophia-Antipolis Méditerranée, France
\(^3\) France Télécom, France
\(^4\) INRIA Rennes Bretagne, France

Abstract. The goal of the MOBIUS project is to develop a Proof Carrying
Code architecture to secure global computers that consist of Java-enabled mobile
devices. In this overview, we present the consumer side of the MOBIUS Proof
Carrying Code infrastructure, for which we have developed formally certified,
executable checkers. We consider wholesale Proof Carrying Code scenarios, in
which a trusted authority verifies the certificate before cryptographically signing
the application. We also discuss retail Proof Carrying Code, where the verification
is performed on the consumer device.

1 Introduction

MOBIUS [BBC\(^+\)06] is a European integrated project developing basic technologies
to ensure reliability and security in global computers formed of a host of Java-enabled
devices, such as phones, PDAs, PCs, which provide a common runtime environment for
a vast number of mobile applications. Its aim is to give users independent guarantees
of the safety and security of mobile applications, using the concept of security through
verifiable evidence emphasized by the Proof Carrying Code (PCC) paradigm [Nec97].
The fundamental view behind PCC is that mobile code components come equipped with
a certificate that can be checked efficiently and independently by the code consumer to
ensure that that downloaded components issued by the producer respects its policy.

PCC complements standard security infrastructures such as PKI, which only guaran-
tee the origin and the integrity of code, and makes an appropriate basis for security of
global computers; however, there remain significant challenges to generalize its use in
security architectures for global computing, in particular:

– Need for comprehensive policies: PCC has mostly been used to enforce safety
properties of applications, including type safety, and memory management safety.
One goal of the MOBIUS project is to show the adequacy of PCC for enforcing
basic security policies such as non-interference and resource control, and for the
verification of functional properties of applications.
– Need for enhanced PCC tools: programming logics and type systems are the two
basic enabling technologies for PCC. However, developing programming logics
and type systems in the context of a full-blown, object-oriented, programming
language such as Java raises a number of challenging issues about efficiency, scalability, and trustworthiness of the PCC infrastructure itself. One goal of the MOBIUS project is to develop efficient and scalable mechanisms to generate and to check certificates, and to prove formally that the security-critical part of the PCC infrastructure is correct.

The purpose of this article is to present intermediate achievements of the project with respect to its goal of achieving efficient and trustworthy PCCs tools. We consider two scenarios, wholesale and retail Proof Carrying Code, and detail for each scenario how to achieve reliable certificate checking.

2 Proof Carrying Code

The purpose of this section is to recall existing approaches to Proof Carrying Code, and to present the main characteristics of the MOBIUS approach.

2.1 Type-Based Proof Carrying Code

The most successful instance and widely deployed application of PCC technology to date, namely the use of stackmaps in lightweight bytecode verification, uses type systems as its enabling technology.

A primer on bytecode verification. Bytecode verification \cite{Ler03} is a central element of the Java security architecture. Its purpose is to check that applets are correctly formed and correctly typed, and that they do not attempt to perform malicious operations during their execution. To this end, the bytecode verifier (BCV) performs a structural analysis and a static analysis of bytecode programs.

The structural analysis checks the absence of basic errors such as calling a method that does not exist, jumping outside the scope of the program, or not respecting modifiers such as final. While simple to enforce, these checks are important and failing to enforce them may open the door to attacks \cite{Gow04}.

The second analysis is a static analysis of the program and is meant to ensure that programs execute in adherence with a set of safety properties, and in particular that values are used with their correct type (to avoid forged pointers). This second pass of bytecode verification is implemented as a data-flow analysis using Kildall’s algorithm \cite{Kil73}. The analysis aims at computing solutions of data-flow equations over a lattice derived from the subtyping relation between JVM types, and from a typed virtual machine which operates on the same principles as the standard JVM except for two crucial differences: the typed virtual machine manipulates types instead of values, and executes one method at a time. In a nutshell, the algorithm manipulates so-called stackmaps that store, for each program point, an abstract state (stack type) that is the least upper bound of the abstract states that have been previously reached at this program point. The stackmap is initialized to the initial state of the method being verified for the first program point, and to a default state for the other program points. One step of execution proceeds by iterating the execution function of the virtual machine over the stackmap. A non-default state is chosen and the result of the execution of the typed