Mobius

Information Society Technologies

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 D4.1

Scenarios for Proof-Carrying Code

Due date of deliverable: 2006-09-01 (T0+12)

Actual submission date: 2006-10-09

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

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

Revision 8093 — Final

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)	

Executive Summary:
Scenarios for Proof-Carrying Code

This document summarizes deliverable D4.1 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`.

The goal of this document is to describe possible extensions of the PCC paradigm which make it applicable to a large number of global computing scenarios, and to highlight the technical challenges that underlie their realization. The traditional PCC scenario, which involves a single code consumer and code producer, does not reflect the complex and distributed nature of global computers, and it is in fact invalidated by several global computing scenarios.

The structure of the document is as follows, after a brief introduction and motivation in Chapter 1, in Chapter 2 we recall the PCC scenarios which exist at the beginning of the project. Then, Chapter 3 discusses important issues for the practical uptake of PCC in the context of global computing, such the translation of source-level properties to compiled-level properties, the combination of static verification and dynamic checking, the different formats used by different flavours of PCC and the implications this can have on the flexibility and efficiency of PCC, and how to have certified certificate checkers. The more speculative part of the deliverable can be found in Chapter 4. It presents a series of scenarios which are likely to be of interest in the general case of global computing. The original PCC scenario is extended in order to consider multiple producers, consumers, intermediaries, external verifiers, and how to effectively handle cases where a certified mobile code is personalized to a particular consumer, where the consumer can customize the policy, or where the certified code is upgraded to a new release. Finally, Chapter 5 presents some conclusions and discusses which of the proposed scenarios and extensions to PCC are subject to ongoing and future work within the context of MOBIUS.

In summary, the document delivers a systematic exploration of the use of PCC in global computing scenarios, and outlines a road map for long term research on PCC. Additionally, we expect that the work performed in MOBIUS in general, and in this task in particular about PCC for global computers, will boost activity in the different research areas involved.

Contents

Chapter 1

Introduction

Proof Carrying Code (PCC) is an emerging technology whose pervasive adoption in global computers requires significant advances. Its founding principle (see Chapter 2), is that incoming components should come equipped with verifiable evidence of their adherence to an appropriate policy that may involve requirements about their safety, security, or functionality. The aim of the MOBIUS project is to contribute to the practical uptake of PCC technologies in global computers. This deliverable summarizes the work performed in Task 4.1, *Scenarios for Proof-Carrying Code* during the first year of the project. Section 1.1 below provides a general overview of what the main technical difficulties for the adoption of PCC in global computing are, whereas Section 1.2 presents the concrete objectives of this task.

1.1 Motivation

While Proof Carrying Code holds the promise of becoming a pivotal technology in next generation security architectures, there are several scientific and technological challenges which need to be solved in order to turn the promise into a reality. To successfully complete the development of a security architecture for global computing, the MOBIUS consortium is confronted with challenges that lie far beyond the current state-of-the-art. The most important challenges are:

1. *Need for comprehensive policies.* So far, PCC has mostly been used to enforce safety properties of applications, including type safety, and memory management safety. It has also been shown adequate for enforcing basic security policies such as non-interference and resource control. However, PCC should also be extended in order to be useful in the verification of functional properties of applications.

2. *Need for enhanced PCC tools.* A fundamental enabling technology for PCC is the use of programming logics. Unfortunately, programming logics for full-blown programming languages raise a number of long-standing problems that include complex features such as aliasing, objects and concurrency, as well as very challenging issues about scalability. A similar conclusion can be drawn for type systems, that constitute another natural enabling technology for PCC. Indeed, existing type systems for safety and security often do not cover the advanced programming constructs pertaining to programming languages for global computers.

3. *Need for a distributed architecture.* Present PCC architectures focus on scenarios involving a single code producer and a single code consumer. In order to apply PCC to global computers in which distributed applications are executed across several devices, we must contemplate scenarios which involve several code consumers. Less obviously, we must also contemplate scenarios where incoming components are synthesized by an automatic process from several code fragments that originate from distinct code producers, or where incoming components may transit before reaching the code consumer through several devices that alter their computational behavior or their data.

It is clear that the adoption of PCC requires both comprehensive policies (Challenge 1) and enhanced PCC tools (Challenge 2). In fact, work packages 2 and 3 in MOBIUS are almost entirely concerned with these two challenges. However, the importance of advanced distributed architectures (Challenge 3) should not be overseen, as without solving this challenge, the impact of PCC would be rather limited, since most of the situations which occur in real-life global computers lie beyond the original PCC scenario. Thus, a central aim of Task 4.1 is precisely to address Challenge 3 (see Chapter 4). In addition, Task 4.1 is also concerned with Challenges 1 and 2 (see Chapters 2 and 3).

In Section 1.2 below we describe in detail the concrete objectives of Task 4.1.

1.2 Task Description

In order to gain wide-spread acceptance, security architectures for global computers should take into account the challenging nature of distributed computation and migrating code that are central aspects of global computers. It is unlikely that all of security issues related to distribution and migration can be addressed by one single security architecture. Nevertheless, it is important to understand, for each security architecture, the extent to which it solves various security issues and the limits to which it can be stretched.

In this deliverable, we develop innovative *scenarios* for applying Proof Carrying Code in the context of global computers because, as already mentioned, research on PCC has so far considered very simple scenarios with a single code producer and a single code consumer, and where the code received by the code consumer matches exactly the code emitted by the code producer. All of these assumptions are questionable in the context of global computing, and there are several scenarios that invalidate them.

For example, components for global computers often involve several code producers. Sometimes these code producers develop fragments of the component individually and later merge them either manually, or through an automated process such as parametrization, instantiation, etc. More often, these code producers successively modify the code: for example, service providers acquire from a software company a generic product that performs some specific service, and customize the software to fit their needs.

Herein we study approaches to PCC that do not rely on the static assumptions of one-producer and one-consumer of code. We examine potential usage scenarios for PCC in the context of global computing and establish conditions under which such scenarios can be realized. Relevant issues that are addressed include:

Presence of several code producers. In the context of global computing it is often the case that the code to be run on a consumer generates from a number of different producers and it is assembled in the code consumer before its execution. In such scenario, who provides the evidence that the full application is innocuous and correct? Where several code producers develop independent fragments of an application, is it possible to construct evidence that the application is correct from the evidence that its fragments are correct? For this, we will study whether existing approaches to modular analysis and verification can be adapted or extended to the context of multiple producers.

Presence of several code consumers. It is also often the case that applications are collectively run by several cooperating devices. In such case, can each device check only the part of the application that it executes? Can we trust the rest of devices involved in an application?

Presence of intermediaries: Sometimes the code has to go through a series of devices (*intermediaries*) from the code producer to the code consumers. These intermediaries may modify both the code and the certificate together. In principle, such intermediates do not need to be trusted since the code consumer will keep the resulting code with the resulting proof. In other situations, intermediaries might actually be part of the consumers trusted code base since certain functions, such as proof checking, might be too expensive for the code consumer and, hence, would be delegated to a trusted proof checker. How should trust of such intermediaries be managed?

Presence of several verifiers. Which situations are better handled, or can only be handled, by off-device PCC in which a code verifier proves the evidence that the code is correct before dispatching it to the code consumers?

The work performed in this task has addressed a series of scenarios. The presentation, much as the results achieved to date, are rather heterogeneous. In some cases the presentation is technical, whereas it remains at an abstract level and is descriptive in other cases. It is important to note that, in order to fully quantify and deploy, several of the scenarios that are presented below require fundamental research in areas such as formal verification, programming languages, and sometimes even in application domains not covered by MOBIUS competences. Furthermore, even for those areas where the MOBIUS consortium has ample competencies to tackle this kind of research, the required effort is outside of the main focus of the project.

1.3 Structure of the Document

The structure of the rest of the document is as follows, in Chapter 2 we recall the PCC scenarios which exist at the beginning of the project. Then, Chapter 3 discusses important issues for the practical uptake of PCC in the context of global computing, such the translation of source-level properties to compiled-level properties, the combination of static verification and dynamic checking, the different formats used by different flavours of PCC and the implications this can have on the flexibility and efficiency of PCC, and how to have certified certificate checkers. The more speculative part of the deliverable can be found in Chapter 4. It presents a series of scenarios which are likely to be of interest in the general case of global computing. The original PCC scenario is extended in order to consider multiple producers, consumers, intermediaries, external verifiers, and how to effectively handle cases where a certified mobile code is personalized to a particular consumer, where the consumer can customize the policy, or where the certified code is upgraded to a new release. Finally, Chapter 5 presents some conclusions and discusses which of the proposed scenarios and extensions to PCC are subject to ongoing and future work within the context of MOBIUS.

Chapter 2

Existing PCC Scenarios

This chapter recalls the existing approaches to Proof Carrying Code (PCC) which will be instrumental for presenting the more advanced scenarios proposed in this document. Three main approaches to PCC are presented. The three approaches have pros and cons and some approaches are more appropriate than others for different contexts. Thus, all of them are considered in MOBIUS. The more traditional, and the more established approaches to date, i.e., type-based PCC and logic-based PCC are described in Sections 2.1 and 2.2, respectively. Finally, Section 2.3 presents abstraction carrying code, a more recent proposal whose main feature is the use of abstract interpretation as enabling technology. This allows the use of powerful and fully automatic techniques for reasoning about a wide range of program properties.

2.1 Type-based PCC

While the original proposal for PCC advocates the use of program logics as enabling technology, see Section 2.2, the most successful instance and widely deployed application of PCC technology to date, namely the use of stackmaps in lightweight bytecode verification for CLDC, uses type systems as its enabling technology. In this section, we briefly review the idea of (lightweight) bytecode verification, and discuss some emerging applications of type systems for the Java Virtual Machine.

2.1.1 Bytecode Verification

Goals Together with stack inspection [68], bytecode verification [106] 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 first analysis, and the simplest, is a structural analysis of the consistency of the class file and its constant pool. During this step, which is described in more detail in Chapter 4 of the JVM specification, one 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 simple checks are important and failing to enforce them may open the door to attacks. E.g. in 2004, A. Gowdiak showed how failure to verify that jump instructions remain within code boundaries allows the construction of a real-life malicious Java midlet application that passes bytecode verification and that could be deployed on Nokia cell phones supporting the KVM.

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, e.g.:

- values are used with their correct type (to avoid forged pointers) and method signatures are respected;
- no frame stack or operand stack underflow or overflow will occur;
- visibility of methods (`private`, `public`, or `protected`) is compatible with their use;

- objects and local variables are initialized before being accessed, see e.g. [69];
- subroutines are correctly used, see e.g. [156, 47].

Ensuring such properties is an important step towards guaranteeing security, and the failure to enforce any of these properties opens the possibility of malicious applets launching attacks.

Besides these properties, the specification of the BCV makes additional requirements that do not directly affect safety of execution, but are more tailored towards an underlying verification algorithm. For example, the specification stipulates that, if a program point can be reached along different paths, the height and type of the operand stacks coincide regardless of the path taken. This rule is enforced by monovariant bytecode verification, which we describe below, but is not enforced by polyvariant bytecode verification, which is more permissive.

Implementation Bytecode verification [106] is implemented as a data-flow analysis of 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.

The data-flow analysis aims at computing solutions of data-flow equations over a lattice derived from the subtyping relation between JVM types. To this end, it uses a generic algorithm due to G. Kildall [98]. In a nutshell, the algorithm manipulates so-called *stackmaps* that store, for each program point, a history structure that represents the program states that have been previously reached at this program point. The history structure 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 states of the history structure. Each non-default state is chosen once and the result of the execution of the typed virtual machine on this state is propagated to its possible successors in the history structure.

Different history structures can be used, reflecting the trade-off between accuracy, efficiency and resource usage that must be made by bytecode verification methods.

- In a monovariant analysis, the history structure stores one program state, which is the least upper bound of the states that have been been previously computed at this program point. In such an analysis, propagating a state in a history structure amounts to taking pointwise the least upper bound (on the type lattice of the virtual machine) of the types appearing in the two states and storing the result back at this location. Termination of the analysis is guaranteed since the set of states does not have infinite ascending chains, and the state stored in the history structure is increasing.

  As noted by R. Stata and M. Abadi [156], collapsing history structures to a single state as done in the monovariant analysis leads to a bytecode verification algorithm that does not handle subroutines as prescribed by the informal specifications of Sun. More precisely, monovariant bytecode verification rejects bytecode programs that make a polymorphic use of subroutines. This use of subroutines can lead to two states, for the same program point, that do not have the same number of local variables or the same number of elements in the operand stack and that would then be merged state into an error state, although the execution is valid.

- In a polyvariant analysis, the history structure stores the *set* of program states that have been previously computed at this program point. In such an analysis, propagating a state in a history structure amounts to adding the newly computed state to the history structure. Termination of the analysis is guaranteed since the set of states is finite, and the size of the history structure is increasing.

  Polyvariant bytecode verification provides an accurate treatment of subroutines, and was introduced independently by P. Brisset (in unpublished work) and by A. Coglio [47]. L. Henrio and B. Serpette [82] propose an improvement of polyvariant bytecode verification in which compatible states in the history structure can be merged so as to keep the size of history structures reasonable. It is interesting that approaching bytecode verification through model-checking [129, 19] results in an analysis which captures a similar class of programs as polyvariant bytecode verification.

2.1.2 Lightweight Bytecode Verification and On-device Verification

In the context of devices with limited resources such as CLDC-enabled devices, applications are verified off-device and, in case of a successful verification, signed and loaded on-device. Such a solution is not optimal in the sense that it leaves a crucial component of the security architecture outside of the perimeter of the device.

In order to remedy this deficiency, there are several proposals for circumscribing the trusted computing base to the device using on-device bytecode verification. One solution adopted in the KVM [46] is to rely on lightweight bytecode verification (LBCV) [143], a variant of byte code verification whose objective, as compared to standard bytecode verification techniques, is to minimize computations by requiring that the program comes equipped with the solution to the dataflow equations. Thus, the role of the lightweight verifier is confined to checking that the solution is correct, which can be performed in one pass. More technically, a certificate in lightweight bytecode verification is a function that attaches a stackmap to each junction point in the program, where a junction point is a program point with more than one predecessor, i.e., a program point where the results of execution potentially need to be merged. Instead of performing the merging, a lightweight bytecode verifier will merely verify that for each program point the stackmap computed by dataflow analysis, using the stackmaps of its predecessors, is compatible with the stackmap provided by the certificate, and continues its computation with the latter. In this way, a lightweight bytecode verifier essentially checks that the candidate fixpoint is indeed a fixpoint, and that it relates suitably to the program. Such a procedure minimizes computations since one just needs to check that the stackmap is indeed a fixpoint. This, as already mentioned, can be done in a single pass over the program, while simultaneously computing a stackmap for program points that are not junction points (it would be possible to attach a stackmap to all program points, but the resulting certificates would be unnecessarily large and require more checks than with a sparser use of annotations). Lightweight bytecode verification is sound and complete with respect to bytecode verification, in the sense that if a program P equipped with a certificate c is accepted by a LBCV, then P is accepted by a BCV, and conversely, if P is accepted by a BCV, then there exists a certificate c (that can be extracted directly from the fixpoint computed by the BCV) such that P equipped with the certificate c is accepted by a LBCV. To date, LBCV is the sole instance of PCC to be widely deployed: currently, LBCV is deployed on all CLDC devices, and it is expected that the use of LBCV will be generalized to all dialects of Java in the future. Figure 2.1 shows the overall Proof Carrying Code architecture and protocol for type-based certificates.

Another solution proposed by X. Leroy [105] to perform on-device bytecode verification on devices with very limited resources such as smart cards, relies on performing a transformation of the code off-device. The goal of the transformation is to produce code that can be verified in one pass without resorting to certificates. In order to achieve the desired effect, the transformation ensures that:

- the operand stack is empty at branching instructions and join points;
- registers have a single type throughout execution.

It is reasonably straightforward to implement a transformation that achieves the desired effects; moreover, the increase in code size and register use is negligible. It is then possible to perform bytecode verification in one pass on transformed code, because there is no merging to be made at junction points.

2.1.3 Beyond Bytecode Verification

The use of type systems in bytecode verification is a consequence of three crucial factors:

Types are intuitive. Types are a particularly simple form of assertions. They can be explained to the user without the need to understand precise details about why and how they are used in order to achieve certain effects;

Types are automatic. While adherence of a program to even the simplest policy is an algorithmically undecidable property, type systems circumvent this obstacle by guaranteeing a safe over-approximation of

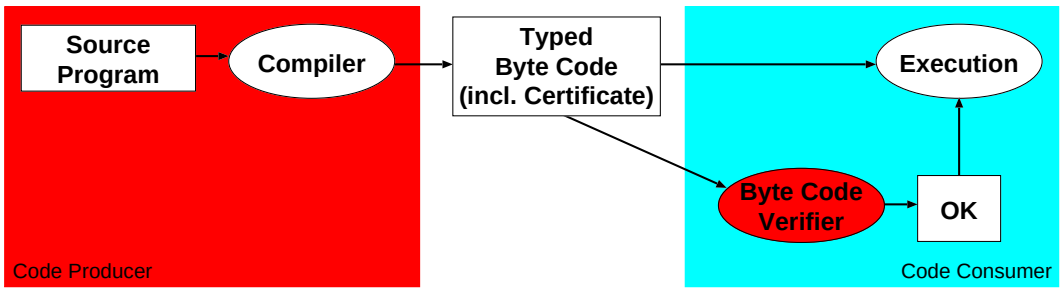


Figure 2.1: PCC architecture and protocol for type-based certificates. Compilers add type information to bytecode, which therefore includes a certificate. The code consumer type checks the code before executing it. Only the type checker (bytecode verifier) is part of the trusted computing base.

the desired policy. For example, a branching statement with one unsafe branch will usually be considered unsafe by a type system. This not only restores decidability but contributes to the aforementioned intuitiveness of type systems;

Types scale up. Besides their simplicity and the possibility to infer types, type systems allow to reduce the verification of a complex system into simpler verification tasks involving smaller parts of the system. Such a compositional approach is a crucial property for making the verification of large, distributed and reconfigurable systems feasible.

It is thus natural to seek to enforce more complex policies using enhanced bytecode verifiers. In fact, a wide scope of properties enforceable by type systems have been carried in the literature. We indicate below some applications that are relevant to MOBIUS.

- Access control and information flow: while the runtime penalty incurred by dynamic access control is acceptable in practice, it is often desirable to detect statically that an application may attempt to violate the access control rules, especially if such attempts result in a security exception that blocks the run-time environment. The problem has been studied both for Java Card [41, 63] and Java [87, 130]. However, access control mechanisms guarantee which principals access but do not guarantee that confidential information will not leak to unauthorized principals [118]. In order to avoid principals (that may access information legitimately) to pass the information unduly, it is desirable to devise security mechanisms that enforce stronger confidentiality policies such as non-interference using information-flow type systems. An information flow type system for a representative fragment of the JVM that includes classes, methods, and exceptions is given in [18];

- Resource control: the MRG project [**?**] has explored the use of advanced type systems to guarantee memory policies for JVM programs generated from a functional language. An efficient alternative to estimate memory usage of Java programs on-device has been proposed by G. Schneider *et al* [38]. Both approaches seem adaptable to a variety of resource usage policies.

- Exception safety: exception safety is an important property that conditions efficient verification of more complex safety and security policies. Existing analyses for JVM programs either deal with specific classes of exceptions [173], or carry a full-blown exception analysis [95]. We believe that developing appropriate lightweight bytecode verification mechanisms for exception safety is a relevant objective for MOBIUS.

- Concurrent programs: there have been many type systems for concurrent fragments of the JVM [64, 103].

However, it is noticeable that advanced type systems for the JVM have remained at the level of prototype implementations, and have not found their way in security architectures. While the situation is partially a

natural consequence of the exploratory nature of some type systems, P. Fong [66] suggests that the situation may also result from a more fundamental limitation of the Java platform, namely that the verification architecture of the JVM is not designed to support extensibility, and advocates the design of extensible protection mechanisms that can be used to accommodate mechanisms tailored towards enforcing application-specific properties such as confidentiality, access control, or resource management. Building on earlier work on proof linking [65], he proposes an architecture that supports pluggable verification modules, and illustrates the principles of his approach by implementing an access control type system as an instance of a pluggable verification module.

2.2 Logic-based PCC

The original PCC infrastructure proposed by Necula and Lee (for lack of a better name, we call this variant Standard Proof Carrying Code, or SPCC), is based on program logics, and more precisely on verification condition generators. In this section, we briefly review the idea of logic-based PCC, and describe how MOBIUS can further enhance existing logic-based technology.

2.2.1 Standard PCC

Standard PCC builds upon several elements: a logic, a verification condition generator, a formal representation of proofs, and a proof checker. Figure 2.2 shows the overall Proof Carrying Code architecture and protocol. We now briefly describe below each element of SPCC:

A formal logic for specifying and verifying policies. The specification language is used to express requirements on the incoming component, and the logic is used to verify that the component meets the expected requirements. SPCC adopts first-order predicate logic as a formalism to both specify and verify the correctness of components, and focuses on safety properties. Thus, requirements are expressed as pre- and post-conditions stating, respectively, conditions to be satisfied by the state before and after a given procedure or function is invoked.

A verification condition generator (VC Generator). The VC Generator produces, for each component and safety policy, a set of proof obligations whose provability will be sufficient to ensure that the component respects the safety policy. SPCC adopts a VC Generator based on programming verification techniques such as Hoare-Floyd logics and weakest precondition calculi, and it requires that components come equipped with extra annotations, e.g., loop invariants that make the generation of verification conditions feasible.

A formal representation of proofs (Certificates). Certificates provide a formal representation of proofs, and are used to convey to the code consumer easy-to-verify evidence that the code it receives is secure. In SPCC, certificates are terms of the lambda calculus, as suggested by the Curry-Howard isomorphism (a.k.a. the proposition-as-types paradigm) [153], and routinely used in modern proof assistants such as Coq and LF.

A proof checker that validates certificates against specifications. The objective of a proof checker is to verify that the certificate does indeed establish the proof obligations generated by the VC Generator. In SPCC, proof checking is reduced to type checking by virtue of the Curry-Howard isomorphism, so that the proof checker verifies that the certificate is of the correct type. One very attractive aspect of this approach is that the proof checker, which forms part of the TCB is particularly simple.

Proof Carrying Code benefits from a number of distinctive features that make it a very appropriate basis for security architectures for global computers. First, Proof Carrying Code is based on verification rather than trust. Indeed, Proof Carrying Code focuses on the behavior of downloaded components rather than on the origin of such components. In particular, it does not require the existence of a global trust infrastructure (although it can be used in combination with cryptographic based trust infrastructures).

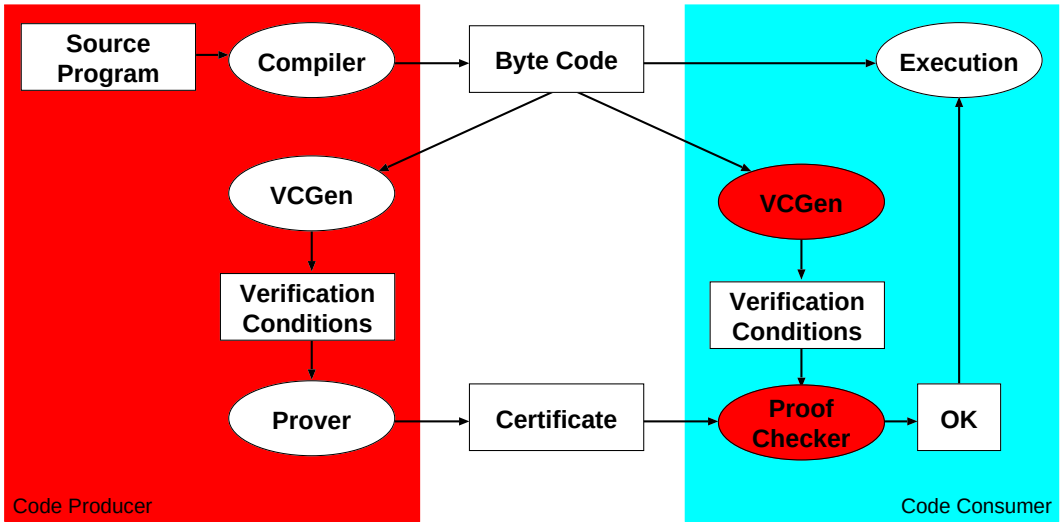


Figure 2.2: Standard Proof Carrying Code architecture and protocol. Systems and data are depicted by ovals and boxes, respectively. The shaded systems are part of the trusted computing base. After compiling the source code, code producers generate verification conditions by applying the VC Generator to the bytecode program. The conditions are then proved by the prover, resulting in a PCC certificate. Like producers, consumers apply the VC Generator to bytecode. The certificate must contain sufficient information to allow the proof checker to discharge these conditions. Only after successful checking, the code is executed.

Second, Proof Carrying Code is transparent for end users. While Proof Carrying Code builds upon ideas from program verification, which in its full generality requires interactive proofs, the PCC architecture does not require the code consumers to build proofs. Rather, it requires code consumers to check proofs, which is fully automatic. Third, the principle of Proof Carrying Code is general; the only restriction on the security policy is that it should be expressible in the formal logic, which is often very expressive. Besides, the basic principles of Proof Carrying Code apply to any logic, as exemplified by recent work on Temporal Proof Carrying Code [25]. Fourth, the Proof Carrying Code architecture is flexible and configurable as it can be used for different policies. In particular, the VC Generator and the proof checker are independent of the policy, while the certifying compiler can in principle be adapted to different safety properties. Finally, Proof Carrying Code technology does not sacrifice performance to security as it advocates for static verification at compile-time, and therefore does not incur in the overhead cost inherent to dynamic techniques based on monitoring.

2.2.2 Certifying Compilers

One fundamental issue to be addressed by any practical deployment of logic-based PCC is the generation of certificates. If logic-based certificates are to be used to verify basic safety properties of code, and it is expected that large classes of programs carry a certificate, then it is important that certificates are generated automatically. Certifying compilers [**?**] extend traditional compilers with a mechanism to generate automatically certificates for sufficiently simple safety properties, exploiting the information available about a program during its compilation to produce a certificate that can be checked by the proof checker. Note that the certifying compiler does not form part of the TCB; nevertheless, it is an essential ingredient of PCC, since it reduces the burden of verification on the code producer side.

Current Instances

The TILT project [162] introduced the idea of certifying compilation, motivating the development of PCC and Typed Assembly Languages. The original purpose of this project was to take advantage of Typed

Intermediate Languages to produce high performance object code from SML. They have shown that much more performance might be gained if type information is propagated to the last stages of the compiler. In particular, they were able to enforce safety policies by type-checking invariants, introducing the notion of certifying analyzer.

An early example of certifying compiler is the Touchstone compiler [**?**], which was intended to explore the feasibility of the pioneer research on PCC. This compiler generates, for programs written in a type-safe fragment of C, a formal proof for type-based safety and memory safety of the resulting program in DEC Alpha assembly language. The Touchstone compiler automatically inserts the loops invariants in the resulting program and generated the correctness proofs. The counter part of this approach is that the ensured properties are restricted to simple properties, namely typing predicates.

Later, Colby *et al* [48] extended this work by transferring type safety check at Java source level to assembly code. They have implemented a compiler that takes a Java bytecode `.class` file and generates an Intel x86 machine code with annotations, together with proof of type safety. First the compiler generates conventional machine code from a `.class` file, properly annotated for a VC Generator. As well as Touchstone, the specification of properties and axioms are expressed in a variant of the Edinburgh Logical Framework and, in this setting, proofs are attempted to be proved automatically. The intention of this work is to determine whether certifying compilers can be applied to programming languages of realistic size and complexity. The source language contains advanced features such as objects, exceptions and floating-point arithmetic and the target language is largely conventional. Although the properties considered are restricted to Java type safety, they mention as future work an extension of the policies to allow enforcement of constraints on resource consumption, such as execution time or memory usage.

The FLINT project [149] aims at building a common back-end for various Higher Order and Typed (HOT) languages. The key idea is based on representing the semantics of several advanced features in a common intermediate language (FLINT). FLINT is a variant of Girard's and Reynolds' polymorphic lambda calculus (Fomega), with a more expressive type system than the Java VM.

2.2.3 Beyond Standard PCC

The use of logic for program verification is a consequence of three crucial factors.

Logic is expressive. During its long development, logic has been designed to allow for greater and greater expressiveness, a trend pushed by philosophers and mathematicians. This trend continues with computer scientists developing still more expressiveness in logic to encompass notions of resources and locality. Today a rich collection of well developed and expressive logics exist for describing computational systems.

Logic is precise. While types generally over-approximate program behavior, logic can be used to provide precise statements of program behavior. Special conditions can be assumed and exceptional behaviors can be described. Via the use of negation and rich quantifier alternation, it is possible to state nearly arbitrary observations about programs and computational systems.

Logic allows to combine analyses. Logic provides a common setting into which the declarative content of a typing judgment or other static analyses can be translated. The results of such analyses can then be placed into a common logic so that conclusions about their combinations can be drawn.

It is thus natural to focus on logic-based PCC infrastructure for certifying Java programs. Relevant issues to be addressed within MOBIUS include the development of an expressive program logic that captures the security requirements identified in WP1. In addition, the program logic should support reasoning about multi-threaded Java applications, and appropriate certificate formats should be defined to allow for compact and efficiently checkable proofs. Finally, it shall be necessary to develop certificate generation procedures that rely on interactive proofs at source level. All of these issues shall be developed in appropriate deliverables; in the forthcoming sections, we briefly examine certificates issues and the generation of certificates from source level proofs.

2.3 Abstract Interpretation and Abstraction Carrying Code

The technique of *Abstract Interpretation* [52] has allowed the development of sophisticated program analyses which are at the same time provably correct and practical. The semantic approximations produced by such analyses have been applied to both program *optimization* and *verification*.

In particular, the application of abstract interpretation to program verification and debugging has received considerable attention. Presenting a survey on the application of abstract interpretation to program verification is out the scope of this document. See for example [51, 78] and their references.

A particularly interesting instance of verification and debugging based on abstract interpretation has been proposed and implemented in the Astrée system [51, 53], and applied very successfully in the Aerospace industry. The use of abstract interpretation for verification, debugging, and run-time testing of assertions has been proposed by members of the Consortium in [36, 84, 137, 86]. The corresponding framework has been implemented in the Ciao multi-paradigm programming system.

2.3.1 Abstract Interpretation-based Verification

For concreteness, we review some basic concepts from abstract interpretation. We consider the important class of semantics referred to as *fixpoint semantics*. In this setting, a (monotonic) semantic operator (which we refer to as S_P) is associated with each program P. This S_P function operates on a semantic domain D which is generally assumed to be a complete lattice or, more generally, a chain-complete partial order. The meaning of the program (which we refer to as $[\![P]\!]$) is defined as the least fixpoint of the S_P operator, i.e., $[\![P]\!] = \mathrm{lfp}(S_P)$. A well-known result is that if S_P is continuous, the least fixpoint is the limit of an iterative process involving at most ω applications of S_P and starting from the bottom element of the lattice.

In the abstract interpretation technique, the program P is interpreted over a non-standard domain called the *abstract* domain D_α which is simpler than the *concrete* domain D. The abstract domain D_α is usually constructed with the objective of computing safe approximations of the semantics of programs, and the semantics w.r.t. this abstract domain, i.e., the *abstract semantics* of the program, is computed (or approximated) by replacing the operators in the program by their abstract counterparts. The abstract domain D_α also has a lattice structure. The concrete and abstract domains are related via a pair of monotonic mappings: *abstraction* $\alpha : D \mapsto D_\alpha$, and *concretization* $\gamma : D_\alpha \mapsto D$, which relate the two domains by a Galois insertion (or a Galois connection) [52].

One of the fundamental results of abstract interpretation is that an abstract semantic operator S_P^α for a program P can be defined which is correct w.r.t. S_P in the sense that $[\![P]\!] \in \gamma(\mathrm{lfp}(S_P^\alpha))$. In addition, if certain conditions hold (e.g., ascending chains are finite in the D_α lattice), then the computation of $\mathrm{lfp}(S_P^\alpha)$ terminates in a finite number of steps. We will denote $\mathrm{lfp}(S_P^\alpha)$, i.e., the result of abstract interpretation for a program P, as $[\![P]\!]_\alpha$.

In most cases, abstract interpretation based verification is concerned with *partial correctness*. This can be interpreted as relating the *actual semantics* of a program P, i.e., $[\![P]\!]$, with the *intended semantics* for the same program (i.e., the specification) $\mathcal{I}$. In this setting, a program P is partially correct iff $[\![P]\!] \subseteq \mathcal{I}$. The key idea in abstract interpretation approach to verification is to actually compute an abstract interpretation $[\![P]\!]_\alpha$ of $[\![P]\!]$ and compare $[\![P]\!]_\alpha$ to the intended semantics. We assume in the discussion that the program specification is given as an abstract semantic value $\mathcal{I}_\alpha \in D_\alpha$. Comparison between actual and intended semantics of the program is most easily done in the same domain, since then the operators on the abstract lattice, that are typically already defined in the analyzer, can be used to perform this comparison. Clearly, a sufficient condition for partial correctness is the following: $[\![P]\!]_\alpha \subseteq \mathcal{I}_\alpha$.

This allows applying the wide range of analysis domains available in the abstract interpretation literature to program verification. These domains vary in accuracy and computational cost and also in the class of properties they observe.

2.3.2 Abstract Interpretation of Java Bytecode

For concreteness, MOBIUS focuses on Java bytecode applications (see Deliverable D1.1). The topic of static analysis of Java bytecode, in general, and abstract interpretation, in particular, has received considerable attention recently. For example, there is significant body of which relies on abstract interpretation techniques in order to reason about the information flow in a program, i.e., the dependencies among program variables. These dependencies are then used to verify that the code does not produce undesired information flows, i.e., it does not leak confidential information (see Deliverables D1.1 and D2.1 for a detailed presentation of Information Flow). Briefly, the security policy is often based on a complete lattice of security classes where information is allowed to flow from variables of a specific security class to variables of higher security classes. In many cases, only two classes are considered for simplicity: High (confidential) and Low (non-confidential). The work on information flow analysis for Java bytecode of [72] first translates the bytecode program into a graph of basic blocks such that the complex control-flow features of the Java bytecode are made explicit. This graph is translated in a second phase into an equation system of Boolean functions such that its least solution approximates the information flow of the original program. Work by [15] also addresses the problem of secure information flow for Java bytecode. This one is based on a static analysis similar to the type-level abstract interpretation used for standard bytecode verification. However, instead of types, the approach uses a domain of secrecy levels assigned to classes, method parameters, and returned values. When a security violation is detected, then the analysis fails. The main difference between these two approaches is that in [72], after analysis, a system of equations collecting the information flow among the different variables is obtained. With such equations, one can study if the program is secure for a safety policy which can be established afterwards. In contrast, in [15] the security domain is fixed a priori, and the Java bytecode is analyzed by using the corresponding inference rules. If the analysis terminates successfully, then the program satisfies the security policy.

A recent static analysis based on abstract interpretation has been introduced for computing relations between the path-length of variables in object oriented programs [155]. By using such relations, the process allows proving automatically the termination of Java bytecode programs dealing with dynamically allocated data structures. The basic idea is to transform the original program by abstractly compiling all its methods. The abstract compilation process generates relational constraints between the formal parameters of the method and the returned value. These constraints refer to the path-length of variables, i.e., to the cardinality of the longest chain of pointers that can be followed from a variable. From such constraints, it is possible to determine (in some cases) if the program terminates by realizing that the path-length of a variable decreases.

Finally, [39] presents a certified algorithm for resource usage analysis aimed at verifying that a program executes in bounded memory. The certification is based on an abstract interpretation framework implemented in the Coq proof assistant which has been used to provide a complete formalization and formal verification of all correctness proofs. The method is formalized as a constraint-based static analysis, implemented as a loop-detecting algorithm which detects methods and instructions that may be executed an unbounded number of times. Concretely, the method detects if the creation of new objects may occur in a loop (including recursion), leading to unbounded memory consumption.

2.3.3 Abstraction Carrying Code

Abstraction Carrying-Code [6] is a general approach to PCC which is based throughout on the use of *abstract interpretation* [52] as enabling technology. The use of abstract interpretation allows ACC to generate certificates which encode complex properties, including traditional safety issues but also resource-related properties like, e.g., the resource consumption the execution of a code is going to require. Moreover, ACC benefits from the maturity and sophistication of the abstract interpretation-based analysis tools available. In ACC, programs are equipped with a certificate which contains an abstraction of the program behaviour.

The certificate (or abstraction of the program) is automatically computed by an *abstract interpretation-based certifier*. Given the set of programs *Prog*, the set of abstract domains *ADom*, the set of abstract safety policies *AInt* and the set of abstractions *ACert*, we define an abstract interpretation-based certifier

as a function certifier : $Prog \times ADom \times AInt \mapsto ACert$ which for a given program $P \in Prog$, an abstract domain $D_\alpha \in ADom$ and a safety policy $I_\alpha \in AInt$ generates a certificate $Cert_\alpha \in ACert$, by using an abstract interpreter for D_α, which entails that P satisfies I_α. In the following, we denote that I_α and $Cert_\alpha$ are specifications given as abstract semantic values of D_α by using the same α. The essential idea in the certification process carried out in ACC is that a fixpoint static analyzer is used to automatically infer an abstract model (or simply *abstraction*) about the mobile code which can then be used to prove that the code is safe w.r.t. the given policy in a straightforward way. The basics for defining the abstract interpretation-based certifiers in ACC are summarized in the following points and Equations.

Abstraction generation. As seen in Section 2.3.1, we consider that the meaning of the program, $[\![P]\!]$, is defined as the least fixed point of the S_P operator, i.e., $[\![P]\!] = \mathrm{lfp}(S_P)$. By using abstract interpretation, we can usually only compute $[\![P]\!]_\alpha$, as $[\![P]\!]_\alpha = \mathrm{lfp}(S_P^\alpha)$. The operator S_P^α is the abstract counterpart of S_P. Then, we define the abstraction of P as follows:

$$\mathsf{analyzer}(P, D_\alpha) = \mathrm{lfp}(S_P^\alpha) = [\![P]\!]_\alpha \tag{2.1}$$

Correctness of analysis ensures that $[\![P]\!]_\alpha$ safely approximates $[\![P]\!]$, i.e., $[\![P]\!] \in \gamma([\![P]\!]_\alpha)$.

Verification Condition. Let $Cert_\alpha$ be a safe approximation of $[\![P]\!]$. If an abstract safety specification I_α can be proved w.r.t. $Cert_\alpha$, then P satisfies the safety policy and $Cert_\alpha$ is a valid certificate:

$$Cert_\alpha \text{ is } \textit{a valid certificate} \text{ for } P \text{ w.r.t. } I_\alpha \text{ if } Cert_\alpha \sqsubseteq I_\alpha \tag{2.2}$$

Certification. Together, equations (2.1) and (2.2) define a certifier which provides program fixpoints, $[\![P]\!]_\alpha$, as certificates which entail a given safety policy, i.e., by taking $Cert_\alpha = [\![P]\!]_\alpha$.

The above certification process carried out by the producer may lead to three different possible status: i) the verification condition is indeed checked and $Cert_\alpha$ is considered a valid abstraction, ii) it is disproved, and thus the certificate is not valid and the code is definitely not safe to run (we should obviously correct the program before continuing the process); iii) it cannot be proved nor disproved. This latter case happens because some properties are undecidable and the analyzer performs approximations in order to always terminate. Therefore, it may not be able to infer precise enough information to verify the conditions. The user can then provide a more refined description of initial calling patterns or choose a different, finer-grained, domain. In both the ii) and iii) cases, the certification process needs to be restarted until achieving a verification condition which meets i).

The second main idea in ACC is that a simple, easy-to-trust abstract interpretation-based checker verifies the validity of the abstraction on the mobile code. The checker is defined as a specialized abstract interpreter whose key characteristic is that it does not need to iterate in order to reach a fixpoint (in contrast to standard analyzers). The basics for defining the abstract interpretation-based checkers in ACC are summarized in the following two points and Equations.

Checking. If a certificate $Cert_\alpha$ is a fixpoint of S_P^α, then $S_P^\alpha(Cert_\alpha) = Cert_\alpha$. Thus, A checker is a function checker : $Prog \times ADom \times ACert \mapsto bool$ which for a program $P \in Prog$, an abstract domain $D_\alpha \in ADom$ and a certificate $Cert_\alpha \in ACert$ checks whether $Cert_\alpha$ is a fixpoint of S_P^α or not:

$$\mathsf{checker}(P, D_\alpha, Cert_\alpha) \textit{ returns true iff } (S_P^\alpha(Cert_\alpha) \equiv Cert_\alpha) \tag{2.3}$$

Verification Condition Regeneration. To retain the safety guarantees, the consumer must regenerate a trustworthy verification condition –Equation 2.2– and use the incoming certificate to test for adherence of the safety policy.

$$P \textit{ is trusted iff } Cert_\alpha \sqsubseteq I_\alpha \tag{2.4}$$

Therefore, the general idea in ACC is that, while analysis –equation (2.1)– is an iterative process, which may traverse (parts of) the abstraction more than once until the fixpoint is reached, checking –equation (2.3)– is guaranteed to be done in a single pass over the abstraction. This characterization of checking ensures that the task performed by the consumers is indeed strictly more efficient than the certification carried out by the producers.

Although ACC, as presented above, is a general proposal not tied to any particular programming paradigm, existing developments [6, 6, 83, 135, 5] are formalized in the context of (Constraint) Logic Programming. This programming paradigm offers a good number of advantages for ACC, an important one being the maturity and sophistication of the analysis tools available for it. It is also a non-trivial case in many ways, including the fact that logic variables and incomplete data structures essentially represent respectively pointers and structures containing pointers.

The verification process in LBV can be formulated in a similar way as in Equation (2.3). Although in this case the single pass of the checker is over the program rather than over the abstraction, as in the ACC framework. Nevertheless, there are two more differences between both approaches. An important one is that the fact that ACC is based on abstract interpretation provides automatic means for certifying a very wide range of properties which brings a greater genericity to the ACC proposal. However, it should be noted that the ACC framework has been developed at the source-level for now, while in existing PCC frameworks the code supplier typically packages the certificate with the *object* code rather than with the *source* code (both are untrusted). LBV is in contrast developed in the context of Java bytecode already.

The main ideas in ACC showed in Equations (2.2) and (2.4) have been used to build a PCC architecture based on *certified* abstract interpretation in [26]. As in ACC, the code producer generates program certificates automatically by relying on an abstract interpreter. The main novelty is that in this method code consumers users proof checkers derived from certified analysers to check certificates. When the consumer does not have the checker, the producer sends the checker together with its soundnesss proof. This soundess proof is then verified automatically by Coq type checker and if the verification succeeds, then the certified checker is installed. This has the important effect of removing the checker from the trusted computing base.

Chapter 3

Certificate Issues and Checkers

Moving the Proof Carrying Code paradigm to the global computer setting imposes many requirements on the nature and structure of certificates and their checkers that were not part of the initial proposal for PCC involving only a single code producer and a single code consumer. In this chapter, we examine explicitly issues involving certificates and their checkers in order to help understand what is known of proofs and certificates now, and how it can be used to support the number of novel scenarios for PCC that we will propose in Chapter 4. First, Section 3.1 studies how to extend PCC to deal with non-trivial security policies and extend the benefits of source-level verification to low-level code by automatically translating certificates from source-level to bytecode level. Section 3.2 presents the case in which, in contrast to PCC, the security policies include properties which are not always statically checkable and how to effectively combine static validation with dynamic checking. Then, Section 3.3 discusses the different formats used by different flavors of PCC. It is important that we have a flexible and rich notion of "proof" and "certificate" in order to meet the many demands for safety and security that are implied by global computing. It is also the case that in recent years, the notion of proof that computer scientists are using in their work has grown to include many new structures which result from using the successful technology of *model checking* and *SAT* (satisfiability) solvers. These two technologies have found wide acceptance in academics and industry, and both of them can yield (in principle) certificates that are not naturally based on the conventional notions of typed λ-terms or proof scripts. Finally, Section 3.4 address the problem of reducing the Trusted Computed Base. We propose to gain confidence in the Proof Carrying Code infrastructure by proving the correctness of the certificate checkers.

3.1 Certificate Translation

Certifying compilation provides a means to automatically generate certificates for basic safety properties. While important in some scenarios, automatic generation of certificates is only possible for a restricted class of properties and programs. For example, it may not be possible to prove automatically basic safety properties for complex software, nor sophisticated policies that involve the functional behavior of relatively simple software. In such situations, one has to use interactive verification environments, possibly combined with automated techniques such as abstract interpretation. These techniques provide a means to guarantee that programs are correct with respect to a formal specification, and are increasingly being used to prove the correctness of safety critical or security sensitive software.

However, interactive verification environments typically operate on source code programs, whereas it is clearly desirable to obtain correctness guarantees for compiled programs, especially in the context of mobile code, where code consumers may not have access to the source program or, even so, they may not trust the compiler. *Certificate translation* aims to provide the benefits of (possibly interactive) source code verification to code consumers, building upon the notion of certificate used in Proof Carrying Code. More precisely, the primary goal of *certificate translation* is to transform certificates of source programs into certificates of compiled programs. Certificate translation shares two important points with certifying compilers: it relies

on the same PCC infrastructure on the consumer side, and certificate translators produce certificates that are checked by the proof checker, and thus do not form part of the Trusted Computing Base. In addition, certificate translation is by design very general and can be used to enforce arbitrary properties on arbitrary programs.

The problem of certificate translation can be expressed informally in a very general form: given a compiler $\mathcal{C}(.)$, a function $\mathcal{C}_{\text{spec}}(.)$ to transform specifications, and certificate checkers (expressed as a ternary relation "c is a certificate that P adheres to ϕ", written $c : P \models \phi$), a certificate translator is a function $\mathcal{C}_{\text{cert}}(.)$ such that for all source-level programs P, policies ϕ, and certificates c,

$$c : P \models \phi \quad \Longrightarrow \mathcal{C}_{\text{cert}}(c) : \mathcal{C}(P) \models \mathcal{C}_{\text{spec}}(\phi)$$

In order to study certificate translation, we must however instantiate the general problem to more specific settings in which it can be tackled. We indicate some dimensions of choice for a PCC architecture below:

- Programming Languages and Compilation Infrastructure: In order to provide an environment where the benefits of interactive verification of source programs are transferred to code consumers, it is necessary to consider at least a three level infrastructure. At the highest level, original source code certificates are built within a verification infrastructure that will strongly depend on the policy language. At the bottom level, final executable code is delivered to consumers together with the transformed certificate. Finally, in the analysis and transformation phases, several unstructured intermediate languages can be considered. This condition depends on the optimizing infrastructure, what kind of properties are expected to be inferred and on the desired optimizations. Nevertheless, focus is put in proof obligations changes entailed by code transformations rather than in how sophisticated are the optimizations applied.

- Verification infrastructure: Certificate translators are tightly bound to the verification infrastructure. In order to get a more compact representation of certificate, it is desirable that work within the MOBIUS project focuses on verification condition generators (VC Generator), which are also used in many interactive verification environments. A VC Generator can be see as an automatic way of applying rules of an Hoare logic (a strategy), so this applications of rules do not need to be stored in the certificate. In this setting, the relation $c : P \models \phi$ is more concretely expressed as a set of certificates c_i representing c, such that each c_i is intended to validate its corresponding proof obligation po_i, each one of those generated by the VC Generator (Figure 3.1). The certificate c is generated interactively

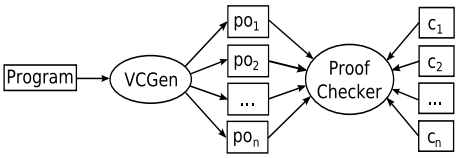


Figure 3.1: Verification Infrastructure

  or automatically transformed from previous phases, depending on which level we are operating on.

- Changes concerning the verification infrastructure. In addition to having to consider languages at different levels of abstraction, as is traditionally inherent in a compiler infrastructure, some changes can be introduced in the verification infrastructure. When specifying a property over a program, the policy language usually refers to state variables that are specific to the execution environment. Since the gradual decrease of abstraction involves both the programming language and the execution environment, considerable changes may be introduced in the policy language and the verification infrastructure. Changes may arise even within the same level of abstraction. For instance, although at the source level a heavy weight VC Generator is used (for instance, JML contains a *modifies* clause

to express *frame properties*), it would be desirable to use a light weight VC Generator to simplify the TCB. This choice does not represent an issue as long as the heavy to light conversion comes equipped with a proper transformation for the certificate. It is important to remark that it is possible to apply this transformation for a simple imperative language, enforcing invariants by means of symbolic execution or strongest postcondition calculus. Finally, at first sight the choice between several proof representations (for instance, lambda terms or proof scripts) affects both size of certificates and the efficiency of the checking process (see Section 3.3 for a detailed discussion on certificate formats). In addition, some proof representations imply that certificates can be more easily manipulated when being translated (e.g., by composition, instantiation or reduction). However, there is some flexibility since this representation is subject to change along the verification-compilation process.

3.1.1 A Road-map to Certificate Translation

A non optimizing compiler generates, from a source program, a sequence of instructions at the intermediate level language. Then, a series of optimizations are performed over the intermediate program. Each of these transformations is applied in an independent step and thus entails translating the certificate.

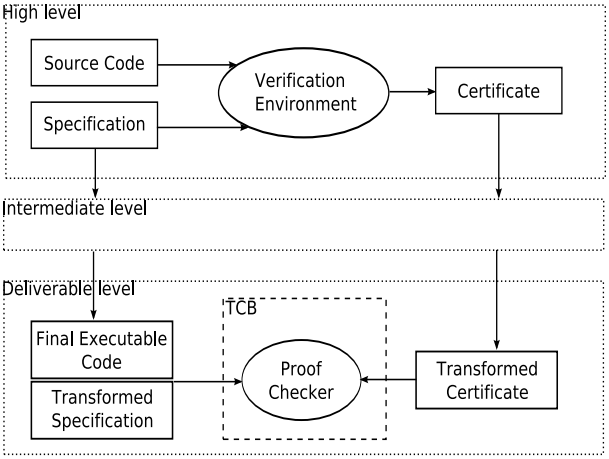


Figure 3.2: A three level infrastructure

Figure 3.2 depicts the overall compilation schema, where the sequence of optimization phases is represented by an opaque box, whereas Figure 3.3 shows the distinction between a non optimizing compiler and a series of independent program optimizations.

A certificate translator for a non-optimizing compiler is relatively simple to define, since compilation preserves proof obligations (up to minor differences). Nevertheless, dealing with optimizations is more challenging. Certificate translation is tightly bound to the optimizations considered and according to the program transformation, proof obligations may be preserved (up to variable renaming, normalization or arithmetic equivalence). However, it may be possible for some optimizations that the VC Generator generates, for the optimized program, proof obligations that are different from the original proof obligations, but equivalent to them up to the results of the analysis. For this reason, certificate translators rely on having previously certified the analysis that justifies the transformation. One must therefore be able to express the results of the analysis in the logic of the PCC architecture, and enhance the analyzer so that it produces a certificate of the analysis for each program.

For the purpose of integrating the proof of correctness of the analysis, optimized programs are augmented with annotations by using the results of the analysis, expressed as an assertion in the logic of the PCC architecture. In addition, depending on whether the optimizations considered are intraprocedural or not, the precondition and postcondition of the optimized function may be modified as well.

There are two approaches to building certifying analyzers: one can either perform the analysis and

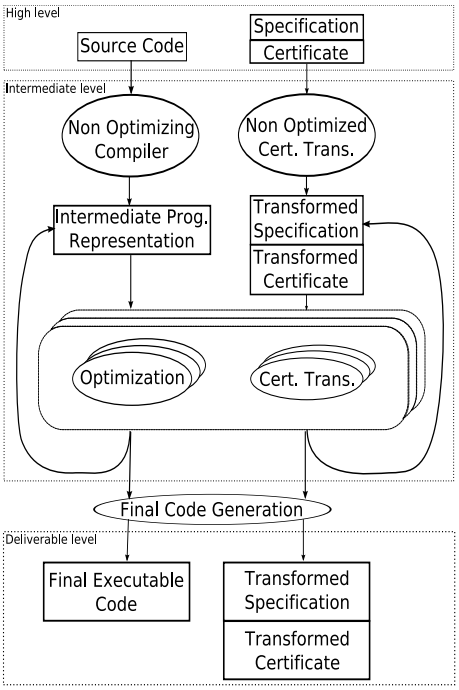


Figure 3.3: Overall picture of the Optimizing Infrastructure

build the certificate simultaneously, or use a standard analysis and use a decision procedure to generate the certificate post-analysis.

Figure 3.4 depicts the process followed by an optimization phase via a certifying analyzer. Three triples are shown (square boxes) consisting on a program p, a specification and a proof that p satisfies its properties. The first group corresponds to the original program, the second one corresponds to the result of the analysis and the remaining one represents the optimized program, with the modified specification and the translated certificate. Certificate translation phases are represented by rounded boxes. A certifying analyzer process takes the result of the analysis RES_A and builds a function f_A, and a proper certificate that f_A satisfies RES_A. Then, an optimizing compiler transforms the program and a certificate translator merges the proof of the analysis with the original proof to build a certificate for the optimized program.

As shown in Figure 3.4, certifying analyzers do not form part of the Trusted Computing Base. In particular, no security threat is caused by applying an erroneous analyzer, or by verifying a program whose assertions are too weak or too strong, or erroneous. In these cases, it will either be impossible to generate the certificate of the analysis, or of the optimized program.

3.1.2 Relation with Certified Compilation and other Techniques

Certificate translation is related to a number of advanced techniques in compilation, which we review below.

Compiler correctness [77] aims at showing that a compiler preserves the semantics of programs. *Certified compilation* [107, 159] advocates the use of a proof assistant for machine-checking the observational equivalence between the input program and the compiler results. In section 2 of [107], Leroy mentions that it is theoretically possible (however, he does not mention it is the proper way) to derive certificate translation from certified compilation. However, the approach is both restrictive and impractical.

Traditionally, compiler correctness results establish a relationship between the inputs and outputs of programs; for example, one standard goal in compiler correctness studies is to show that if running program p on an initial configuration c returns some final result v, then running the corresponding compiled program $\mathcal{C}(p)$ on the same initial configuration c shall also return the same final result v. Under suitable conditions, one can also prove the converse, i.e. if executing $\mathcal{C}(p)$ with initial configuration c yields the final value

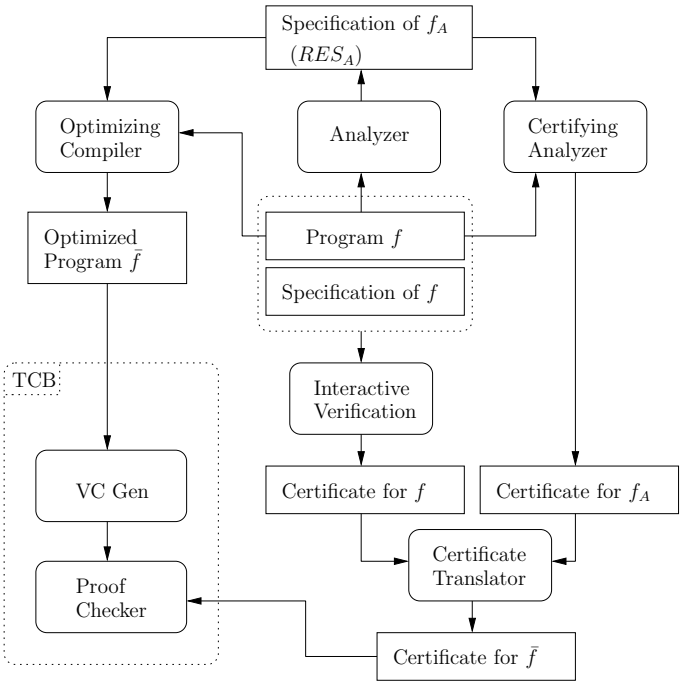


Figure 3.4: Overall picture of certificate translation

v, then executing p with initial configuration c yields the final value v. It can easily be observed that this implication may be exploited to transfer evidence from source code programs to compiled programs. Indeed, assume that we want to prove for some program p that $R(c, v)$ holds whenever executing $\mathcal{C}(p)$ with initial configuration c yields the final value v. By the above implication, it is sufficient to show that $R(c, v)$ holds whenever executing p with initial configuration c yields the final value v, which is exactly what the certificate of the source program p should establish.

Thus, it is in principle possible to build certificate translators from certified compilers. There are, however, some major drawbacks. The approach is impractical because certificates encapsulate the definition of the compiler and its correctness proof on the one hand, and the source code and its certificate on the other hand. Checking certificates of compiled programs is more costly than the combination of checking the correctness of the compiler, checking the certificate of the source program, and compiling the source program. Thus, certificates are outrageously large, and outrageously costly to check.

On the other hand, the approach is restrictive since with the above notion of certified compiler, properties are confined to conditions about the input/output behavior of programs. But unfortunately, many interesting properties of programs must be specified using assertions or ghost variables. A further difficulty with this approach is that it requires that the source code be accessible to the code consumer, which is in general not the case. For similar reasons, it is not appropriate to take as certificates of optimized programs pairs that consist of a certificate for the unoptimized program and of a proof that the optimizations are semantics preserving.

Another related line of work is translation validation, proposed by A. Pnueli, M. Siegel and E. Singerman [128], and credible compilation, proposed by M. Rinard [141]. The latter aims at showing, for each individual run of the compiler, that the resulting target program implements correctly the source program, i.e. it has the same semantics. This is achieved by the automatic verification of invariants representing the result of the analysis (for every program point in the source code). These results are then used in each program point to justify the semantic preservation of the observable effects from the source to the transformed program. This technique does not allow to verify that a given specification is satisfied. Related work has also been done by X. Rival [142, **?**], who uses abstract interpretation techniques to infer invariants at the source level and compile these invariants for the target level.

3.2 Combining Static Verification and Dynamic Checking

In mobile, ubiquitous computing environments, we need to achieve functional computing environments in the presence of many forms of mobility of users, hardware and software. This often demands close integration of both dynamic checking and static verification methods. A typical example of static verification is Java bytecode verification and its enhancement to security, as well as various model checking techniques. A basic example of dynamic checking is access control, where identification of a principal who requests an action, checking the corresponding policy for that principal, and enforcing the policy, are all done at run-time. Such access control is hard to carry out statically since, for example, the policy itself can change over time. Consider for example, that as a result of a security breach on a web server, one needs to tighten the access control policy inside an organization: this temporarily changes the security policy, and the security monitor now functions dynamically reflecting this new policy. On the other hand, most of type safety in bytecode can be verified statically at loading time, and some properties such as secure information flow are known to be hard, or even impossible, to guarantee by means of run-time checking only. Note that type safety is a basic requirement for safe usage of pointers, which is fundamental for other security properties (one of the well-known methods to illegally obtain high-level clearance is through the use of corrupted pointers). Thus, combining static validation and dynamic checking is often essential for guaranteeing practical security. Dynamic verification can also involve distributed coordination of authentication and validation of policy.

Another example is the use of program logics (where program logics may be used for validating properties both statically and dynamically) to guarantee a safety of mobile code through PCC with a sophisticated local access control policy. Such a framework would be relevant to one of the scenarios discussed in the S3MS Project [134] where a traveler needs to use mobile code in a foreign city she is visiting. First, a vendor (provider) of the downloadable code equips the code with an assertion written in an access control logic and its proof. This assertion will specify which components and resources this code would access. Second, when this code is downloaded, a local browser checks if this assertion satisfies an access policy using a proof checker. Once this has been verified, the code will run inside an instrumented protection domain which guarantees its safe behavior by assuming its statically guaranteed properties as well as enforcing its dynamically checked properties. That is, some aspects of its access behavior are delegated to dynamic checking on the basis of its statically guaranteed properties. For example, static checking will guarantee that the code does not leak information and that it always checks access levels for a class of resources and aborts if its privilege does not reach them: however access to certain critical resources is checked dynamically.

At a more concrete level, there are several language technologies needed to realize this idea. In the context of Java, Polymer [20] is an expressive language framework which allows simple specification of run-time policy on security properties in Java. They studied different ways in which run-time behavior is altered depending on concerned safety properties. By combining their framework with a validation framework such as the one proposed by Fong [66, 65] (in which class loading and bytecode verification are separated, cf. Section 2.1.3), we shall obtain a basis for realizing the combined static/dynamic validation. Again for Java, Chander and others [43] presented an interesting scheme where the guarantee given by a simple form of dynamic validation (represented by simple annotations) is subjected to static checking by a user. By restricting to simple resource usage policies, they can make use of tractable program annotations and a simple predicate transformer calculus for validation. They experimented their scheme with a standard archival program written in Java. The use of annotations in this work can be a useful technique in other contexts. These examples suggest effective integration of static and dynamic validation techniques.

Another concrete example of combined dynamic checking and static verification arises in the context of abstract interpretation-based verification, discussed in Section 2.3. The fact that undecidable properties are involved and safe approximations used in the reasoning means that it will not always be possible to achieve full static verification. A solution in these cases, when the property is amenable to be checked at run time, is to resort to dynamic checking by including the appropriate run-time tests in the program. This way, a program which is not statically guaranteed to satisfy the security property is still allowed to execute, but its execution is *monitored* in such a way that any possible violation of the security policy will be detected at run-time before it actually takes place.

Recently, leveraging the results of static checking to increase the efficiency of dynamic checking has been discussed for tools that understand the Java Modeling Language [37]. Practically, rich specifications impose significant overhead during runtime checking. For example, executing an application that uses a few model classes and rich specifications that use finitely quantified first-order expressions in invariants can slow execution speed dramatically. If one were to prove that certain specifications were always valid, then during runtime checking those assertions could be elided. This integration of static and runtime checking is a planned feature of the MOBIUS tool, developed in Task 3.6.

3.3 Certificate Size, Certification Complexity, Efficiency Trade-offs

Logic has played a significant role in computer science for years. For example, logical formulas have been used for a wide variety of purposes, ranging from type systems, to formal specifications of computation systems, to circuit design, to artificial intelligence and natural language semantic. Along with logic formulas, the related notions of provability and satisfiability have had important roles to play, as can be witnessed by the frequent use made of theorem proving systems such as Coq and Isabelle and of model checkers and SAT solvers by computer scientists. One can argue, however, that it was with the introduction of PCC that *proofs-as-objects* became important to computer science. Proofs as formal structures have been studied for decades by logicians and philosophers, such as Gentzen, Prawitz, and Girard, to name a few. Their study has provided a number of mature and sophisticated approaches to represent proofs so that one can formally manipulate them: for example, normalization and cut-elimination provide rich ways to compute with proofs. The results of that theoretical research have been exploited in a number of automatic and interactive theorem provers where proofs can be stored, manipulated, re-played, etc. But the work in PCC appears to be one of the first efforts by people working outside of the theorem proving community to actually use proofs as part of a computer system who primary purpose was not the generation of proofs themselves. Of course, computer scientists probably have a rather different list of requirements concerning proofs-as-objects than, say, philosophers. For example, the identity of proofs, a focus on early work on natural deduction [131], seems completely unimportant for PCC. Also, formal manipulation of proof object has played little role in PCC, although it is likely to play more of a role as this topic matures, for example with the addition to the PCC scenario of the program manipulation techniques discussed in Sections 4.5 and 4.7.

Within PCC, many engineering-related concerns are paramount: we shall concern ourselves with the following four properties.

1. Since certificates need to be stored, communicated, and checked, their *size* is particularly important. If certificates are large, communicating them from the code producer to the code consumer can dominate communication costs. If certificates are small, computationally intensive search might need to be performed in order to reconstruct details of the implied proof. Throughout this section, we address the size of certificates in only high-level and qualitative terms: there currently is not enough experimental evidence and theoretical understanding to provide *fully formal* ways relate certificate sizes to certificate formats.

2. Checking that a certificate actually contains a proof of the appropriate safety property, requires using the *computational resources* of both memory and computation time. Some certificate formats require that a certificate is completely loaded prior to it being checked, which implies that memory usage must accommodate certificate size. Some certificates are, instead, *stream-based*, in which case the certificate can be checked as it is read and, as a result, it does not need to be stored.

3. Certificate checkers as software components can range from simple and compact to sophisticated and multi-functional. Their *complexity* is tied to other aspects of certificates. Simple certificate checkers are likely to be based on certificate formats based on simple, inference-rule-level proof structures and, as such, are likely to require large certificates. On the other hand, certificate checkers can, instead, contain many specialized checkers for a number of specialized domains. As a result, certificates matched with such checkers can be smaller since they need only mention the use of a specialized proof checker

and not detail the proof structure itself. Since ultimately the correctness of a certificate checker must be established, hopefully, a simple and compact prover will likely be easier to formally prove correct than a more sophisticated checker.

4. The *high-level structure* of certificates as artifacts can play an important part of a certificate's role in PCC. It is possible that in many situations, a certificate for an applet is generated by a compiler and used in exactly one way: it is transmitted with the binary code and checked against that code. If the certificate has no other role to play, then there is no need for it to have structural properties others than those required by the compiler and checker. It is also likely, however, that certificates need to be stored and to survive changes in versions to both the compiler and the checker, as in the component update scenario presented in Section 4.7. It is also likely that novel PCC architectures can do interesting things with factoring proof checking if, in fact, certificates can be factored and specialized, as already discussed in Section 4.5. If such formal manipulation of certificates is important, then the high-level structure of certificates must be studied carefully.

The literature on PCC discusses essentially four different kinds of certificate formats. We shall refer to them using the following names: typed λ-terms, proof scripts, oracle strings, and fixed points. We shall now examine each of these formats in more detail.

3.3.1 Certificates as λ-terms

The original proposal for proof carrying code [122, 121] used certificates that were encoded directly as dependently typed λ-calculus [79]. Basing certificates on typed λ-calculus allows PCC to exploit a large and mature literature on the theory and implementation of certificates. For example, a great deal is known about the high-level structure of typed λ-calculus: proofs can easily be specialized via substitutions (via, for example, β-reduction) and their structure can be checked statically for a wide range of properties. Certificates based on λ-terms hold great promise as a setting in which some certificates can be made generic and, for example, stored in libraries from where they can be retrieved by compilers and then specialized (see Section 4.5). Given their symbolic nature, it is also likely that they can be used for representing certificates that are more remote from particular low-level technological considerations. For example, certificates based on dependently typed λ-terms allows one to first specify a logic [14] in which to conduct a proof: even the choice of logic can, in principle, be incorporated into certificates and not fixed ahead of time.

The complexity of a checking proper typing of λ-terms has been well studied. In particular, there is a rich literature describing ways to effectively implement certificate checkers in the style of syntax-directed type checking. The most naive approach to using typed λ-terms fills proof objects with complete details and stores many of those details in redundant fashions. As a result, checking certificates can be done without search and by simple-to-design checkers.

Unfortunately, the flexibilities of using typed λ-terms as certificates does come with significant costs. In particular, various uses of this approach to PCC has been characterized by a serious problem with respect to the size of certificates: in particular, certificates are generally huge and can easily be an order of magnitude larger than the code for which they are associated. Some improvement in size has been made by refining the kinds of λ-terms actually built in a dependent typed calculus [**?**] or by explicitly tailoring their structure using a programming language, such λProlog [9], that supports direct computation on λ-terms. While some of this research has allowed reducing the size of typed λ-terms, it seems likely that certificates based on this kind of representations will be too large for at least a wide variety of applications. While typed λ-terms will almost certainly remain the touchstone reference for certificate formats, it seems that other certificate formats must also be considered in the hope of achieving more compact proof representations. As mentioned above, other verification technology yield proof-like objects that are not naturally considered as λ-terms. Integrating model checking, for example, into PCC seems to require examining other forms of certificates as well.

3.3.2 Certificates as Proof Scripts

Theorem provers are often guided to proofs by scripts. Such scripts started out as records of all the interactive commands that a user issued to guide a theorem prover to a proof. Although proof scripts have evolved to be more structured objects, they still provide the essential function of guiding a particular prover to a proof of a particular formula. Since proof scripts can be stored and replayed, they can be seen as providing certificates, since they are all that is needed to find a proof in a given theorem prover. The MRG project (see [**?**]) has experimented with using proof scripts in the role certificate in this fashion.

Since proof scripts were originally designed to store particular commands for particular interactive theorem provers, such as Isabelle and Coq, proof scripts will generally only function with that one, particular theorem prover: both the proof producer and proof checker need to be the same prover. Being closely attached to a particular prover also makes it possible for the proof script to refer to high-level, sophisticated routines available within that prover, which can reduce the size of the certificate.

Anchoring proof scripts to a particular prover and, more specifically, to sophisticated and computationally intensive parts of such provers, can significantly reduce the usefulness of proof scripting as certificates in PCC, because it is not feasible to include an entire theorem prover, such as Isabelle or Coq, on mobile devices. On the other hand, if scripts were restricted to simple inference rules, it should be possible for such scripts to be understood by a range of simple and unsophisticated checkers that mobile devices could include. Of course, such restrictions to simple inference rules can mean that proof scripts are likely to yield larger certificates. See [12] for a proposal on how proof scripts might be treated more abstractly and, hence, more flexibly.

Independently of their size and their use of simple or sophisticated commands (inference rules), proof scripts can, in principle, be processed in a stream-based fashion. Thus, proof scripts can be checked as they are communicated and they do not need to be stored in their entirety within the proof checker. Such processing can significantly reduce the memory requirements of a proof checker.

Given the current practice, proof scripts are rather fragile: simple changes in the formulas to be proved or small changes to the version of the theorem prover can invalidate a proof script. Such fragility might be improved if certificates are based on a reduced and more universal set of inference rules. Given the possibly close connections that proof scripts should have with certificates based on typed λ-terms, one might expect that proof scripts can have similar useful high-level properties.

3.3.3 Certificates as Oracle Strings

A rather striking departure from the above two symbolic approaches to representing proofs is given by the *oracle string* approach presented by Necula in [123]. In that approach, a non-deterministic theorem prover is given the task of proving a theorem: whenever the prover has to pick from n choices, it reads $\lceil \log_2 n \rceil$ bits from an oracle string to resolve that choice. As a result, the oracle is used to drive the theorem prover to a final proof without search, and as such, the oracle string can be considered a proof. Experimental evidence suggests that oracle strings can be significantly smaller than proofs based on typed λ-terms. As with proof scripts, checking an oracle string can be done in a stream-based fashion, so memory demands for the proof checking device can be low.

While the small size and low cost of checking an oracle string are appealing, a potential problem with them is that there are no currently known ways to manipulate or compose them. Thus, oracle strings for subprograms might be hard to use directly when trying to find certificates for larger programs (oracle strings are based on guiding the search for cut-free proofs). They are also fragile in the sense mentioned for proof scripts: small changes in the formula to be proved or in the version of the theorem prover can invalidate an oracle string. It might be possible to provide oracle strings with some useful high-level properties so that they are less fragile and more modular. Proof theory classifies non-determinism into "don't care" and "don't know" (called *asynchronous* and *synchronous* [7]): such formal results might go a long way to provide some structure to oracles so that they can be composed in meaningful ways.

3.3.4 Certificates as Fixed Points

As described above in Section 2.3.3, certificates in the Abstraction Carrying Code (ACC) approach to PCC use a fixed point in an abstract model as a certificate. Basing certificates on (abstract) models instead of proofs certainly provides interesting new dimensions to apply not only the mature field of abstract interpretation to PCC but also the general area of model checking.

A possible connection between model checking and proof can be seen in the following simple example: a safety policy for a mobile device might require that its various resources can only be accessed in certain fashions (for example, once an applet accesses a password file to do authentication, the applet cannot make a network connection for fear of making public the password file). Such a safety policy can be specified by insisting that the applet's communication behavior is *simulated* by some given process calculus expression. Simulation in this context is based on a (greatest) fixed point that can be determined using standard model checking software or by ACC. Such fixed points can also be seen as describing an actual sequent calculus proof in a suitable logic [113]. The connection is related to the above mentioned distinctions between "don't care" and "don't known" non-determinism. In particular, if one takes out of the sequent calculus proof of simulation all of the "don't care" non-determinism, then the resulting structure essentially contains a collection of key formulas and terms that were required to know in order to complete a proof. Distilling down to exactly this information reduces the sequent calculus proof to a particular set of formulas that can be seen as a simulation fixed point. Conversely, one can also move from a fixed point back to a proof by inserting the required "don't care" non-determinism that is needed for building a sequent calculus proof.

While proving a safety property in ACC requires iteratively computing a fixed point, that fixed point (the certificate) can be checked by a single scan of the fixed point. Also, ACC seems to allow for capturing symbolic and expressive properties using certificates of compact size and checkers that can be of low sophistication while being general [6].

In early work on PCC, Peter Lee [104] has asked how model checking advances could be useful added to PCC, since model checkers have become very strong at automatically proving simple theorems about low-level code. The main problem with using model checkers for PCC is the generation of appropriate certificates. Fixed points as certificates might well serve to make this connection possible.

3.3.5 Richness of Certificate Formats

Clearly there is a richness to the possible structure of certificates. Just as there has not been just one programming language (nor just one programming paradigm: cf, object-oriented, imperative, functional, logic) to solve all programming problems, it is not likely that there will be just one approach to certificates, particularly given the various engineering demands they must meet in a PCC deployment on a global computer.

The richness of certificate formats outlined above implies that the notions of proofs and certificates should be able to accommodate a wide variety of engineering concerns and that the theory of proofs has come a great distance since its early days as a topic of symbolic and philosophical logic.

Of course, an effort like that implied by the MOBIUS project may well need to pick initially a particular certificate format that is flexible enough to capture a wide range of envisioned applications and for which there exists tools and expertise among the consortium members. The diversity above implies, however, that the proof carrying code effort more broadly will have a great deal of flexibility available to it as different sets of applications are considered and as technology for proving advance, say, from traditional theorem proving to model checking to, say, winning strategies in game-based modeling.

3.4 Certified Certificate Checkers

In principle, PCC enables the code receiver to check security properties using a small trusted verifier. However, when dealing with realistic programming languages, the certificate checker becomes a non-trivial large program. For example, in the case of the Touchstone verifier, the VC Generator represents several

thousand lines of C code, having as consequence that "there were errors in that code that escaped the thorough testing of the infrastructure" [124]. This problem has led to several proposition to reduce the size of the Trusted Computed Base (TCB) of a PCC architecture. We shall now examine the different proposed approaches.

3.4.1 Foundational Proof Carrying Code

The *foundational proof carrying code* (FPCC) of Appel and Felty [10, 8] gives stronger semantic foundations to PCC. In this approach, the code producer gives a direct proof that, in some "foundational" higher-order logic, the code respects a given security policy. Compared with Standard Proof Carrying Code, this approach corresponds to directly embedding the correctness proof of the program analysis into the safety proof. Thus, with this technique, the VC Generator is removed entirely, and the TCB is minimalist.

3.4.2 Direct Verification of a VC Generator

Instead of simply removing the VC Generator, Wildmoser and Nipkow [171, 170] prove the soundness of a *weakest precondition* calculus for a reasonable subset of Java bytecode. Hence, they formally proved the correctness of a core element of a VC Generator, which can then be excluded from the TCB. Necula and Schneck [124] also extend a small trusted core VC Generator and describe the protocol that the untrusted verifier must follow in interactions with the trusted infrastructure.

3.4.3 Certified Program Logic

The *Mobile Resource Guarantee* (MRG) project [24, 13] has produced a PCC infrastructure with minimal TCB for proving properties related to the resource consumption of a code with explicit memory management. To reason about intermediate code annotated with memory consumption information, they build an intermediate layer of customized inference rules from a generic program logic. The soundness of this logic is checked in Isabelle. Certificate checking is then reduced to checking a proof in this dedicated logic.

3.4.4 Certified Abstract Interpretation

Certificate checkers can also be obtained by construction. Certified abstract interpretation is a technique for extracting a static analyzer from the constructive proof of its correctness: since the Coq proof assistant allows extracting the computational content of a constructive proof, a Caml implementation can be extracted from a proof of existence, for any program, of a correct approximation of the concrete program semantics.

Using certified abstract interpretation has the following three advantages:

- the PCC infrastructure has semantic foundations as strong as those of FPCC;
- code certificates can be built from results of untrusted static analyzers without the need for re-proving these results inside a theorem prover;
- proof carrying proof checkers can be built in such a way that the code consumer can check their correctness using a type checking mechanism.

The third point opens up for the possibility of safely downloading proof checkers, adding flexibility to the PCC infrastructure. This direction has been followed by Besson, Jensen and Pichardie in [26] who develop a certified analysis for array out-of-bounds accesses and extract an optimized and certified certificate checker.

3.4.5 Certified Certificate Checkers in the MOBIUS project

All the previous approaches advocate a formal verification of the certificate checker. In the MOBIUS project we will fit this approach in a same framework. Part of this infrastructure is detailed in deliverable 3.1 [117] and is summarized in Figure 3.5.

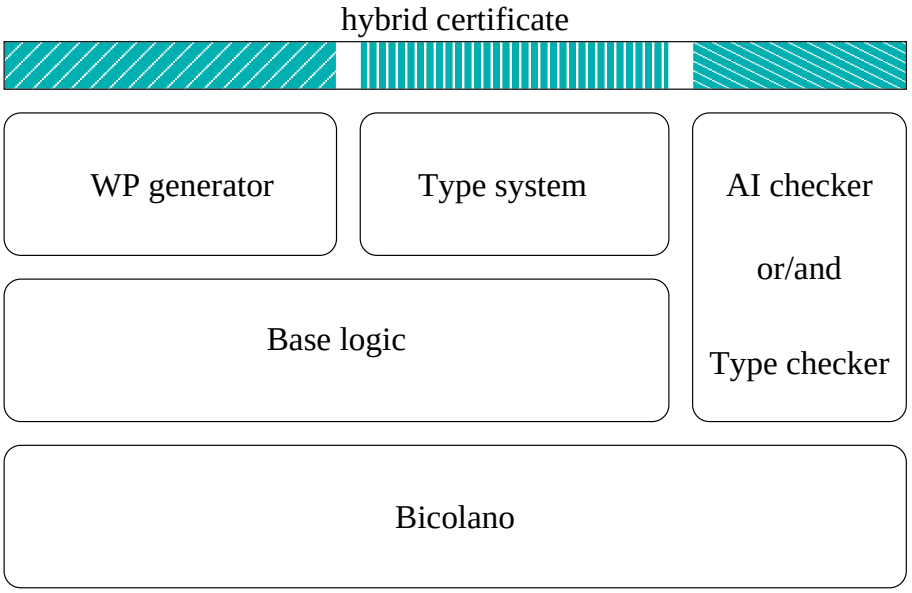


Figure 3.5: Certified certificate checker infrastructure

At the bottom is the Coq formal specification of the JVM, named Bicolano. The base logic is then proven correct with respects to Bicolano. Derivation in this logic can either be obtained by the WP generator or an advanced type system (as one used in MRG project [24]). Both components are formally proven correct with respect to the base logic. Certificate checkers for abstraction carrying-code or advanced type checking (for example, for information flow) can be also directly proved correct with respect to Bicolano. Such an infrastructure will hence support hybrid certificates composed of different kind of certificates. The computational capabilities of the Coq system support the creation of foundational certificates, directly expressed in terms of a Bicolano (or base logic) security property. The figure does not mention possible interactions between certificate checkers: the WP generator can, for example, benefit from the information provided by an advanced typed system in order to automatically discharge some proof obligations. Such scenario for hybrid certificates will be investigated in Task 4.2.

Chapter 4

Global Computing Scenarios

The distributed and heterogeneous nature of global computers calls for the development of PCC technologies that go beyond the initial scenario with one code producer and one code consumer, and where code installation on the consumer side is a one-time process. Indeed, global computers are designed to provide execution support for distributed applications which are themselves built from components that originate from several code producers that are transferred to several code consumers, possibly through intermediaries that—perhaps maliciously—modify the components, and possibly the process is repeated for different software upgrades.

The purpose of this section is, therefore, to consider scenarios that involve multiple intermediaries: producers, consumers, and verifiers, scenarios where the code can be *personalized* to a particular consumer, where the consumer can locally customize the safety policy, and where the code is subject to repeated upgrades after being installed. Also, we will aim at identifying possible bottlenecks in adapting PCC techniques to these scenarios. For the sake of clarity, we consider the features above in isolation. However, naturally, global computing scenarios tend to combine these features. For example, one can envision a component-based (and thus built from many producers) distributed application to be offered to consumers via an intermediary that delegates verification (i.e. certificate checking) to several certification entities, and upon successful completion of certificate checking, makes it available to consumers that will execute it on different nodes. Less obviously, the features are not necessarily orthogonal: for example, it is conceivable that intermediaries perform just part of the checking and leave other parts of the checking to the consumer, which is a special case of multiple verifiers. We discuss one such scenario in Section 4.6, where the intermediary performs partial verification to let consumers perform on-device verification of customized policies. The more general problem of intermediaries is discussed in Section 4.1: one central issue here is whether or not the intermediary is to be trusted and, if it is, how to combine trust and verifiable evidence. Section 4.2 considers issues that arise when applications are assembled from multiple producers: the central issue here is to support modular verification, so that the code of each component can be certified in isolation, and the global application can subsequently be verified without re-checking the certificate of each component. As a generalization of this problem, we consider in Section 4.7 the problem of building certificates incrementally in order to make provisions within the PCC architecture for components upgrade: a modular architecture is, of course, a prime importance in global computing. Then, we consider in Section 4.3 the problem of verifying a distributed application by focusing on making sure that a computation performed remotely on an untrusted grid is indeed correct. In Section 4.4 we consider the case where we have generic code which is then personalized to a given consumer. This scenario is of particular interest in the context of global computing since a large number of devices with different features may co-exist. Finally, in the context of global computing it is not realistic to expect that all consumers will be able to check all possible certificates. Thus, in Section 4.5 we consider a scenario which involves the existence of multiple verifiers which allow performing off-device checking of (part of) the security policy.

Overall, the situations considered in this chapter provide a good indication of the potential and difficulties of scaling PCC technologies to global computing scenarios. Our intention here is to contribute to a long-term

road map for PCC research by describing situations that arise naturally in the context of global computers, but for which the use of PCC technology is currently speculative. Developing appropriate technologies for all the situations considered is not possible within the scope of MOBIUS, since some of these situations require either radical evolution in the current software industry or technical research advances that are go beyond the expertise of this consortium.

4.1 Trusted Intermediaries

In the setting of Proof Carrying Code, we shall use the term "intermediary" to refer to some computer or computational resource that is involved in the transaction associated to getting code from a code producer to the code consumer. In a simplified view of PCC, what is called "retail" PCC in Section 5.2.5 of Deliverable 1.1, there may be no intermediaries involved: code and certificates simply move over a network (usually not considered an intermediary) from the code producer to the code consumer. Alternatively, proofs may be checked by a trusted third party (what we call "wholesale PCC") who then signs the application, in which case proofs are not downloaded to the code consumer.

In a more involved and realistic scenario for Proof Carrying Code, various kinds of intermediate computational resources might be needed. For example, various kinds of libraries might be needed containing, for example, proofs of the correctness of different software. Since the needs of proof checking might overwhelm a resource limited machine such as a telephone, actual proof checking could be moved from the telephone to intermediate machines with more computing power. As we shall see below, various kinds of additional intermediaries can be considered.

A fundamental problem with admitting intermediaries into the PCC setting is that we must now know whether or not we can trust or need to trust an intermediary. If the intermediary helps in constructing or transforming the mobile code and its safety proof (say, by specializing generic proof objects or by replacing one component by a safe but functionally different component), then that intermediary does not, in fact, need to be trusted: if the safety proof still matches the mobile code (even after processing by a malicious intermediary), the code can be considered safe and can be consumed. If, however, the intermediary is checking a proof of correctness so that the code consumer does not need to do so, then the code consumer must have a strong degree of trust in the proof checking service.

Let us consider this issue of off-loading proof checking from the code-consuming devices to a network of certificate checking servers. The servers function would be to check certificates and then assure the code consumer that certificates are, in fact, correct. Presumably, what the certificate server needs to send to the code consumer is much smaller and less sophisticated than the actual PCC certificate.

Off loading of certificate checking from mobile, limited resource devices to specially designed servers that exist to check certificates might be desirable for many reasons.

For example, computationally limited devices, such as telephones, might not be able to check anything but the simplest certificates and, as a result, the certificates they check are likely to be only the weakest ones. Moving certificate checking to servers should improve network behavior: if code producers are only required to submit their full certificates to certificate checking servers (of which there should be a small number) and not to all code consumers (of which there should be a much larger number), then network traffic should be reduced. In addition, servers would be able to cache repeated requests for checking the same code+certificate bundles. Also, more capable servers should make it possible to have more sophisticated and possibly larger certificates checked: in such a case, higher levels of safety and security of downloaded code could be achieved. Of course, relying on centralized servers can expose the system to denial-of-service attacks: an attacker could try to overload the proof-checker server.

Also, PCC requires certain infrastructure commitments: storage and cycles for the certificates, upgrading of checkers and associated software, etc. While PCC should make devices more secure, it also adds a degree of complexity to them to achieve that security. Makers of global computers may want to move some of that complexity off of smaller devices and adopt a more server-client model of certificate checking. Also, certificate checking is not a frequently required task in contrast to, say, the main operations of code consumers (e.g.,

database operations, video output, VOIP, etc), so optimizing limited computing resources away from PCC infrastructure will make sense for small devices.

Moving certificate checking to servers means that the code consumer must trust explicitly the certificate checking servers. Thus, some infrastructure for managing trust must be adopted. We now consider how to integrate PCC with trust architectures by considering the questions: How can one use trust to customize the security policy of PCC? How can trust between host devices affect verification tasks in PCC for distributed applications? Should the notion of trust be integrated within the logic used for PCC itself?

Trademarks and published certificates

Before considering more sophisticated and explicit models of trust below, consider the following simple scheme for trust that relies on the fact that certificates are ultimately based on proofs and logic and that these are publicly available formal systems. Thus, one way to manage trust might be for certificates to be simply posted by the code producer in their entirety on some public web page. The code consumer might be willing to take evidence of the certificate being made public as enough reason to trust the certificate. Why? Given that certificates are public, other agents on the web (run by, say, consumer-based or government-run agencies) could be downloading certificates and checking them. If any incorrect certificates are found, the company responsible for publishing the incorrect certificate could lose significant respect and market share. Thus, companies with trademarks to protect (for example, France Telecom, Google, and VeriSign) might be expected to take great care with any certificates they sign and, hence, such certificates might be trusted.

Checking certificates offline

Assume that distributor D of programs and certificates wishes to have its programs run on a collection of devices $\mathcal{C}$. One way that this might be achieved is to designate another machine P that confirms that the alleged proof is a proper proof of the safety of the associated program. This requires, of course, that the devices in $\mathcal{C}$ trust P and that cryptographic signatures are available for P to sign off on correctness of the certificates. Such a signature scheme is, of course, open to the possibility that, if the proof checker P has an error, or is compromised by an attacker, some devices in $\mathcal{C}$ can be lead into executing unsafe code. Also, such signature mechanisms are vulnerable to spoofing: see, for example, the case reported as Microsoft Security Bulletin MS01-017 (titled "Erroneous VeriSign-Issued Digital Certificates Pose Spoofing Hazard"). Spoofing attacks might also be managed differently in this setting since, if there is doubt about the truth of a given signed message, one can go back to the original program and certificate issued by D and have them checked and signed again.

There might be situations, however, where such negative aspects of an offline checking scenario are outweighed by the fact that the devices in $\mathcal{C}$, which might be computationally restricted, do not need to check proofs and that proofs do not need to be sent. If the checker P and the devices $\mathcal{C}$ are all part of a similar authority (such as a phone company), it might be felt that the trust infrastructure between them can be managed without problems of spoofing and erroneous provers.

Trusting a theorem prover without a proof

There may be situations in which a device is willing to trust a particular theorem prover or compiler without actually having a proof. In such a situation, the device would presumably need to receive a formula packaged along with the code, with a cryptographic signature for the package as a whole. The device would then need to check that (a) the signature is in a list of provers that it trusts and (2) the formula that is asserted to be a theorem is, in fact, the required safety condition for the interface(s) that the associated code purports to implement.

A theorem prover might achieve this status of being trusted in at least a couple of ways. A prover might be an open source project in which the code is inspected, tested, and used by many individuals over a long period of time. Trustworthiness might be granted by some people based on long-term familiarity with the code and its operation. On the other hand, the prover might have a brand name associated to it, in which

case, a corporation might have a financial incentive to ensure its correctness. A bug in a theorem prover can, of course, be catastrophic: if a prover can prove $0 = 1$ then it can prove anything. In the area of trust management, brand names, such as VeriSign, serve a related purpose.

PCC and Software Certification Entities

Code inspection by software certification entities is a technique that is currently used to certify code as suitable for mobile devices, in the absence of formal verification. Code inspection could still be used for legacy components, and for components that would be too expensive to interactively verify.

This means, however, that a signature from a software certification body could be treated by the system as something that establishes that proof obligations have been fulfilled, even though it does not guarantee such thing in reality. As a compromise, non-critical properties of a component could be verified by inspection, and critical properties by formal verification, resulting in a *generalized hybrid certificate*.

With such generalized, or (partially) manually-generated certificates, it is important not to blindly rely on the properties checked by human inspection, in machine-checked proofs. Instead, proof checkers need explicit policies specifying when such evidence should be trusted.

Trust Management

Levels of trust can also be used to lower the bar that a component has to pass in order to be accepted. If a component is signed as coming from a highly trusted source, or is being used by a trusted principal, it might, for those reasons, be permitted to access protected resources and send data from those resources to other hosts. The proof obligations placed on that component might therefore be less restrictive (though not necessarily simpler) than those of an untrusted or less-trusted component. This technique could potentially be useful to remove burdensome requirements from trusted developers.

For example, a software development team may consider that trusting each other, and their build machines, not to introduce malicious code is enough, and the components they are building should only be required to prove that they implement their API contracts and stay within resource bounds. Other properties, such as security properties, could be omitted for such trusted components, where security is non-critical. Conversely, a developer might decide that certain security properties are essential, but full API correctness is too expensive to verify for the benefits obtained.

Such decisions about the proof obligations to impose for trusted components could be made by (in the most likely scenario) client or "base" component developers, by system administrators, or by users. These decisions, perspectives, and negotiations of trust must be encoded in the certificate in a formal manner, so that devices can automatically understand the trust context of a given certificate check. Logics of trust [109, 110] are used to reason about these situations. To encode such terms, typed property-value pairs, with an accompanying ontology of trust [93] are the only structures that are necessary. *Generalized certificates* contain such structured, ontological metadata. Thus, *generalized hybrid certificates* are certificates that combine object code, formal properties of that code, logic and type-based specifications and proofs, and structured metadata about trust.

In systems that use public key infrastructures [67] (PKIs), *revoking* a key (encoded in a PKI certificate) is as or more important than issuing and maintaining the key in the first place. A key is revoked in many circumstances: the owner has lost the private key or the key's passphrase, has gone out of business or is otherwise defunct, a product associated with the key is no longer supported, a more sophisticated encryption algorithm is introduced, etc.

Revoking a key in a PCC setting has new consequences, especially in light of the above scenarios (e.g., in Section 4.1). Different revocation intentions have radically different impact on the validity of PCC infrastructure entities and PCC certificates themselves. For example, the fact that a prover is no longer supported by a given company does not suddenly mean that all proofs generated or checked by that prover, or that indirectly depend upon that prover's behavior in some fashion, are suddenly invalid. Likewise, if

a private key has been compromised, this does not mean that all proofs signed with that key are invalid because they can still be checked by the code consumer.

Given these scenarios, it seems necessary to formally encode *revocation intention* of keys in PCC infrastructures that rely upon any kind of trust infrastructure [108]. Current work on such encodings is ad hoc. There is no consistency between different applications (e.g., while PGP and GPG have the same encoding, OpenSSL only resembles them [125]). Such an encoding is not complex in realization, but must be carefully expressed, in perhaps both mathematical and legal terms, and extensible to new reasons for revocation.

Non-PKI systems frequently rely upon Webs of Trust [97, 93] to encode the trust relationships discussed in this section. By signing *keys*, one can indicate relationships between keys, between key owners, and between the other entities that keys represent.

4.2 Multiple Producers

Software systems are more varied and complicated today than at any point in the history of computing. There are hundreds of languages, tools, and technologies to choose from for developing software systems in any given application domain. The field of software engineering research has similar variety, as there are an enormous number of formalisms, methodologies, and approaches.

Concretely, projects consisting of millions of lines of program code, written in a dozen languages, and developed and maintained by hundreds of people across multiple companies are now commonplace. Ensuring the reliability and security of such software systems is an important goal for future research, and partly the subject of Hoare Grand Challenge on the Verifying Compiler [88]. The adoption of PCC technology would radically contribute to this goal, as PCC considers proving properties of programs as natural part of software development, and puts proofs on an equal footing with the programs whose correctness they establish.

In the following sections, we examine the main issues in extending PCC technology to make it appropriate for such software systems: the development of software by several producers, and the use of multiple languages and formalisms to develop and validate the code.

4.2.1 PCC for Component-based Software Engineering

Component-based software engineering (CBSE) is a relatively young topic that aims at facilitating the development and maintenance of software systems through the integration of off-the-shelf components [160].

The main considerations unique to CBSE are:

component identification is necessary because one must identify which components to use to fulfill the requirements of the decomposed system we wish to construct. The identification of components is accomplished to date mainly through *non-technical* means, primarily through developer familiarity with component developers, companies, and communities. Rarely are components described, even in an informal fashion.

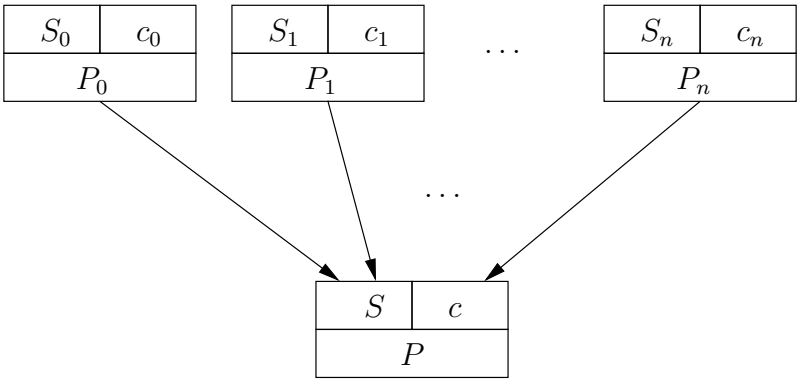
component development is different because creating components is about describing and writing *reusable* and *refinable* pieces of software [132, 27, 28].

component refinement is necessary because very few components exactly match the identified requirements [35].

component integration is the final critical aspect because only through the integration of components can one produce a software system fulfilling the identified requirements and having the desired properties.

The integration of traditional PCC in a component-based software development process impacts on two steps and restricts its application to a single programming language.

The first step that is influenced is developing certified components which is achieved using standard PCC techniques. The second step is certifying the software systems themselves. Validating component-based software requires the use of compositional verification techniques, so that not only the code but also its certificate are reused.



Suppose that the program P is assembled from components $P_0 \ldots P_n$ provided by different producers $cp_0 \ldots cp_n$; each component comes equipped with a specification S_i and a certificate c_i. In order to certify that P verifies some specification S, one needs a method to produce a certificate c that P satisfies S from the certificates of each component. Assuming that each component is a set of Java classes and methods, with P_0 being the main component and S being S_0, and that specifications consist of pre- and post-conditions as well as invariants, we fall back on standard verification techniques.

Figure 4.1: Multiple producers

In other words, we need compositional methods to derive certified properties of the software system from certified properties of its components. Compositional verification is a challenging subject that remains actively researched, however Figure 4.1 illustrates that in some common scenarios standard verification technology is sufficient. (The topic of reasoning about modular programs in the context of abstract interpretation-based verification is discussed, e.g., in Pietrzak et al [127].)

In addition to these two steps, it may be necessary to transform certificates for components that must undergo adaptations prior to integration. This subject is further developed in Section 4.5.

Integrating PCC in component-based software development has two main advantages: first, it increases software reliability and security, which is the goal of PCC. Second, component specifications (against which certificates are checked) provide very useful information to decide whether components qualify for a certain task [111, 112]. Furthermore, the range of specifications that are considered within PCC permits a more extensive qualification process whereby not only functional properties of components are considered, but also resource consumption and security properties such as confidentiality and integrity of data.

4.2.2 PCC for Multi-Languages Software and Multi-Logics Certificates

The limitations of traditional PCC, particularly its focus on a single programming language, is problematic in a modern development context. By relying instead on a common PCC *platform*, next generation CBSE will be PCC-enabled without this limitation. Additionally, in the future, some of the reasons for choosing to use multiple programming languages and technologies are complemented by the ability to choose amongst multiple logics and verification techniques.

Modern multi-language system development is realized in one of several ways:

library-based development where the primary unit of reuse is a small-to-medium sized library, typically written in the C programming language. C is the lingua franca of libraries, thus it is the dominant language of reuse today. But because C programs are so difficult to verify, there is little hope of seeing PCC platforms that contain traditional C libraries.

object-oriented classes where sets of related classes are collected into reusable units like packages or frameworks [29]. OO frameworks have been the focus of much verification research over the past

decade, thus they are the main place that we see reuse of verified "proto-components" happening today in the pre-PCC development space.

Unfortunately, nearly all OO languages and runtimes, until the advent of the Java VM and Microsoft's CLR, did not support inter-language interoperability, and only Microsoft's effort was intended to support multiple languages by design. Also, the fundamental paradigm adopted by most OO languages, that of class-as-types (as seen in Java, C++, C#, Eiffel, etc.), is widely considered to be a flawed theoretical foundation in academic circles [140].

CORBA-based systems was the first-generation, widely-adopted technology for integrating subsystems written in multiple languages. Subsystems are described using an Interface Definition Language (IDL) that specifies, in a programming language-independent fashion, the type interfaces of the subsystems. Some research exists in specifying more than the type interfaces of CORBA subsystems, e.g., the work of Sivilotti and Zinky [150, 151, 178], but such work has not been widely adopted in the CORBA community outside of occasional lightweight use of OCL [169].

web services are the latest craze in multi-language system integration. Essentially, web services are not critically different than CORBA in nearly all ways.

Initial work in component identification in modern multi-language development environments focused on informal component descriptions using natural language, structured keyword-based descriptions called facets, and module interconnection languages [133, 11]. These approaches were shown to be unscalable without the incorporation of a significant amount of tool and organizational support [157, 174, 157, 57].

The idea of generalized certificates were inspired by these challenges and were first used in multi-language, mobile, distributed component-based systems like the Infospheres Infrastructure [44].

Components in this frameworks, called *djinns*, had human readable and machine parsable metadata attached to them [75]. This structured metadata included such information as: the author(s) of the component, the corporate entity supporting the component, the component's version number, the URLs from which the component's source or binary might be downloaded, and a summary of the component's purpose.

Because of the use of human-readable, structured descriptions and URLs, components could be discovered and evaluated through the use of standard web technologies like search engines [99]. Unfortunately, these descriptions did not contain some of the essential ingredients necessary for a PCC platform, namely formal specifications of component properties.

Other groups focused on integrating formal methods into CBSE. Cramer et al [54] summarize some of the early methods including algebraic specifications, the Π component-based description language, and transformational systems. Early work on formal specifications of component properties focused less on behavioral properties and more on formalizing the aforementioned informal properties (a la early certificates) and resources [165]. Finally, once these pieces were in place reuse-centric software development methods were proposed [172], sometimes with tool support.

The research tool support created to date, but which has never left the lab, has come in two main forms: reuse repositories that experiment with different kinds of (usually informal) specification matching [60] (though rarely formal specifications were used [73]) and knowledge-based environments [58].

4.3 Multiple Consumers

Global computers provide support for remote or distributed execution of algorithms that cannot be executed on a single node, either because the algorithm is computation-intensive, or because the node has restricted resources. These scenarios have generally not been contemplated by work on PCC (excepted by the Concert project [50]), and indeed PCC is not immediately applicable to all such scenarios.

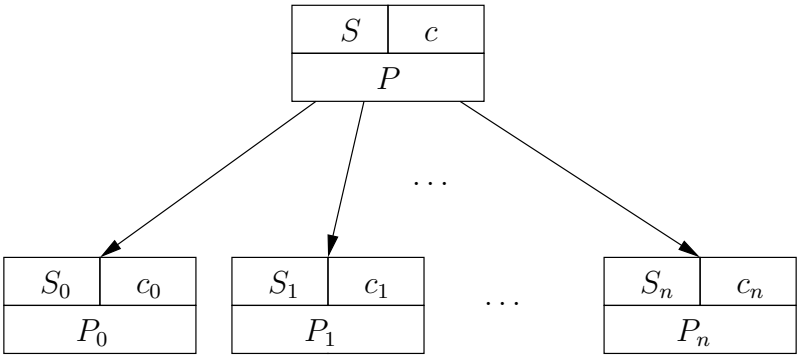


Figure 4.2: Multiple consumers

4.3.1 Distributed Computations among Trusted Hosts

Ensuring security or correctness of distributed programs is greatly facilitated if distributed computation takes place among trusted hosts. Here the program P is executed among nodes which trust each other, so that ensuring that the distributed execution of P is secure or correct can be reduced to proving the security or correctness of each distributed fragment. Suppose, as in Figure 4.2, that the program P is partitioned into fragments $P_0 \ \ldots \ P_n$ to be executed on different hosts $h_0 \ \ldots \ h_n$, with P_0 being the "main" fragment of the program (we consider a static distribution of code for clarity, but the argument can be extended to evolving sets of nodes).

Functional correctness: in order to guarantee that the overall program behaves as expected, it is sufficient that each fragment comes equipped with a specification S_i and a certificate c_i that can be checked locally on h_i (or delegated to trusted intermediaries), and that the specification S_0 of P_0 ensures the overall program correctness. Assuming that programs are written in Java and each fragment of the program corresponds to a set of classes with its methods, we fall back on standard verification techniques;

Security: in distributed scenarios, confidentiality and integrity are the main issues. In order to guarantee both properties, one must rely on network security, e.g. that data travels encrypted and signed so that confidential data cannot be leaked during communication over the network or that tainted data can be detected. One must also ensure that each program fragment guarantees confidentiality and integrity in isolation, which can be achieved by using program logics or type systems. One interesting approach to enforce confidentiality/integrity without necessarily trusting all hosts is secure program partitioning [177], which divides a program to prevent information leaks or integrity violations. Program partitioning can be tuned to reflect the level of trust one has in the different nodes, e.g. so that critical program fragments are directed towards reliable hosts whereas non-critical fragments are sent to an arbitrary host.

While the above considerations are primarily directed towards genuinely distributed applications, a promising trend of operating systems research consists in exploiting the benefits of partitioning on a single node [144].

4.3.2 Distributed Computations among Untrusted Hosts

For many scenarios, including grid computing, the assumption of a trust relationship between different hosts must be abandoned. Thus, a user U asking a remote node[1] to perform some computation c with some set

[1] In this scenario R has to trust the code sent to it by U. Since R can protect itself using a traditional PCC architecture, we focus here on the security of U

of values $\vec{v}$ on his behalf cannot presume upon reception of the result r that R is returning the result of c, nor that R has been performing c with the correct set $\vec{v}$ of values, nor even that R has been using c at all! In order to ensure that r is correct, the only solution for U is to go through independent validation of r. Since U cannot in general perform the computations independently or relying only on trusted hosts, because the resources to perform the computations on site or via trusted hosts are not available, it must rely on a *checker* that verifies the correctness of the result, and that can be performed on site or via trusted hosts. In simple instances, no additional information is required to check the correctness of the result efficiently (e.g. when the algorithm is computing a solution to a system of equations), but in other instances one needs additional information in order to verify efficiently that the result is indeed correct.

While PCC is not directly targeted towards ensuring that the results of remote computations are correct, one can adapt ideas from PCC and develop a variant of PCC, which we call Proof-Carrying Result, that can be used to secure distributed computations in global computers.

Proof Carrying Result Proof Carrying Result (PCR) is an approach which requires that the result comes equipped with a certificate which provides certain evidence that the computation is correct; certificates typically consist of a combination of data (additional witnesses that are produced during the computation and used to build the result) and proofs (that establish some expected property of the witnesses, or of the final result). Proof Carrying Result inherits many benefits from PCC:

PCR is based on verification rather than trust. Indeed, PCR focuses on mathematical properties of the result rather than on its origins. In particular, it does not require the existence of a global trust infrastructure (although as for PCC it can be used in combination with cryptographic based trust infrastructures).

PCR is transparent for end users. While PCR uses certifying algorithms, which may be difficult to program or consuming to run, PCR requires code consumers only to check certificates, which is fully automatic, and not to build these certificates;

PCR is general, flexible and configurable. Formal frameworks for certificates, e.g. type theory, are very expressive, and lend themselves to efficient verification, thus PCR is in principle applicable to a wide range of algorithms. Furthermore, it is possible to specialize certificates and certificate checkers for each particular algorithm.

PCR is resource-aware. Indeed, PCR technology advocates for succinct certificates that can be checked efficiently and aims at avoiding performing costly computations.

Formally, given a function $f \in A \to B$ and an entry $a \in A$, the computation of $f(a)$ is delegated to an untrusted party. The result is returned and verified by the user. To do this, the user must be equipped with a function $\text{check}_f \in A \times B \to \text{bool}$ such that $\forall (a, b) \in A \times B, \text{check}_f(a, b) = \text{true} \Rightarrow b = f(a)$. A broader perspective of PCR deals with finding b such that $R(a, b)$, where the relation R holds the specification to a certain problem. This is, we generalize the functional specification f to a general input-output specification R.

Proof Carrying Result serves similar purposes than probabilistic result checking proposed by M. Blum and S. Kannan [31]. However, both approaches fundamentally differ at a technical level: whereas PCR may require additional information to ensure correctness, probabilistic result checking emphasizes probabilistic verification and does not require additional information.

In general, PCR allows the untrusted part to provide additional data H intended to ease the checking process. Thus, one may have a checker function $\text{check}_R \in A \times B \times H \to \text{bool}$ such that $\text{check}_R(a, b, h) \Rightarrow R(a, b)$. This means that no matter what value of h is provided, the check will fail unless $R(a, b)$ is true. The untrusted party is in charge of providing an appropriate h to convince the user that b is an appropriate answer. As we shall see, the existence and construction of such an h is at the core of efficient result certification algorithms.

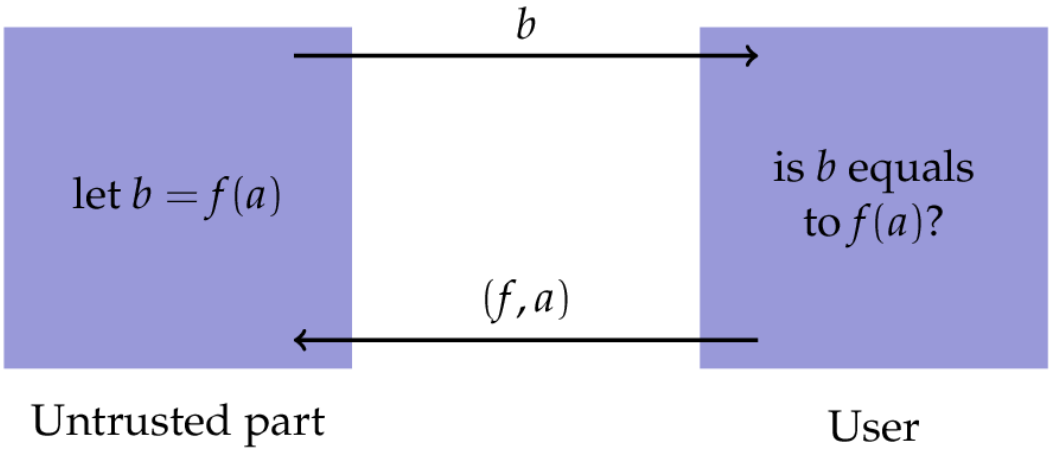


Figure 4.3: Result Checking Scheme

One can think of the additional hints as responsible for selecting a successful trace in a non-deterministic solver program. Thus, the complexity of checkers for a problem is the same as that of a non-deterministic solver where hint reads have been transformed to nondeterministic reads. In the case of decision problems in NP, a polynomially checkable certificate may be given for positive answers. A decision problem is in co-NP iff negative answers have polynomially checkable certificates. Thus the set of decision problems for which polynomial checker may be given is exactly NP $\cap$ co-NP [114].

The task of designing such a checker program and an oracle (result producer) providing well defined hints to eliminate non-deterministic choices seems to be highly creative. In what follows we show some examples of certifying algorithms and data structures that illustrate the challenge posed by this approach.

Challenge 1: certifying algorithms The development of proof carrying result is conditioned by an evolution in the design of algorithms in scientific disciplines and fields that rely on software-intensive computations that must be executed over Grids. In order to make PCR a practical tool for secure remote computations, the discipline of certifying algorithms must become prevalent. More concretely, one needs to develop for each algorithm A a compact format for certificates, typically a data structure used to store auxiliary values that can be used to check the result of a run of A efficiently, as well as a means to generate certificates and check them efficiently. Devising such certifying algorithms is a fundamental challenge for PCR.

One basic example of certifying algorithm is the extended GCD (Greatest Common Denominator) algorithm: although GCD does not allow a straightforward checker, extended GCD allows a simple and efficient checker, and is itself not much harder to compute.

$$GCD(x, y) = d \text{ where } d|x \wedge d|y \wedge \left(\forall d', d'|x \wedge d'|y \Rightarrow d'|d\right)$$

The extended GCD algorithm further computes u and v such that $d = ux + vy$. Such a u and v constitute the certificate; in this case, the certificate always exists and implies the minimality of d, i.e. they guarantee that the quantified part of the specification is true.

One may notice that it is simple to obtain d once u and v have been provided. This shows that in a result certification framework, the hint or certificate may become more important than the actual result. A plain result provided by an untrusted party has no value, whereas a certificate does.

In some more advanced cases, certificates may not always exist or may be harder to compute. In [74], B. Grégoire, L. Théry and B. Werner build efficient primality checkers using Pocklington's criterion, which provides a set of sufficient conditions that must be verified by some partial prime decomposition of $P - 1$ to ensure that the number P is prime.

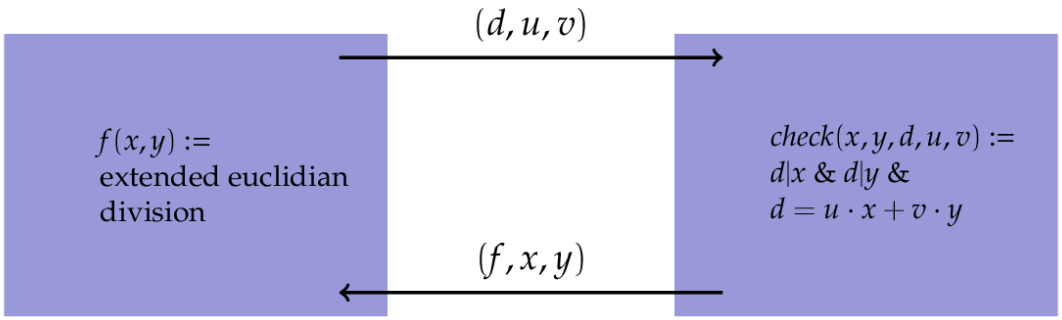


Figure 4.4: Extended GCD

Pocklington's criterion *Given a natural number $n > 1$, a witness a, and some pairs $(p_1, \alpha_1), \ldots, (p_k, \alpha_k)$, it is sufficient for n to be prime that the following conditions hold:*

$$p_1 \ldots p_k \quad \text{are} \quad \text{prime numbers} \tag{4.0}$$
$$(p_1^{\alpha_1} \ldots p_k^{\alpha_k}) \quad | \quad (n-1) \tag{4.1}$$
$$a^{n-1} \quad = \quad 1 (\equiv n) \tag{4.2}$$
$$\forall i \in \{1, \ldots, k\} \ \gcd(a^{\frac{n-1}{p_i}} - 1, n) \quad = \quad 1 \tag{4.3}$$
$$p_1^{\alpha_1} \ldots p_k^{\alpha_k} \quad > \quad \sqrt{n}. \tag{4.4}$$

From the point of view of PCR, there are two simple but central observations to make:

- given n, it requires much more computation power to determine suitable numbers $a, p_1, \alpha_1, \ldots, q_k, \alpha_k$, which constitute a *Pocklington certificate*, than to check that these numbers verify the conditions 1-4 above;
- checking primality of a natural number n with certificate $p_1, \ldots, p_k$ and a boils down to
  1. conditions 1-4 can be checked by purely numerical computations. Verification of condition 3, may make further use of PCR infrastructure by requiring an extended GCD certificate to be provided.
  2. verification of condition 0 can be done recursively. Note, however, that Pocklington's criterion cannot prove that 2 is prime, so that another form of certificate is required for 2 (it can be no certificate at all, since 2 is trivially prime).

There are other theorems that suggest certificate formats and checkers for primality, notably in relation with elliptic curve cryptography; however these certificates are usually difficult to build. Algorithms to check primality are interesting in the context of PCR and secure distributed computations, in particular for deploying cryptography on devices whose resources are so restricted that they must rely on external devices, e.g. to generate random prime numbers.

Although the idea of *certifying algorithms* is not widely spread, there are several interesting examples of certifying algorithms beyond primality. Indeed, early approaches to result checking recognized that it may be useful to return, together with the actual result being calculated, additional hints that enable the result to be verified by a simple and efficient checker—whereas the lack of those hints would only leave space for a checker that is as complex or inefficient as the original program. For example, Melhorn [114] identifies functions providing these hints as having extended interfaces or providing certified results, and a number of other examples can be found in the literature. We point out some of them in the following non-exhaustive list with the intention of illustrating application domains of the idea.

- Solving linear equations (i.e. $Ax = b$ for x). If a solution exists, x can be provided. Insolubility can be certified by providing c such that $cA = 0$ and $cb \neq 0$.

- Checking whether a polynomial $P(x_1, \ldots, x_n)$ is non negative is in general a difficult problem. However, a sufficient condition is being able to decompose it into the sum of squares (SOS). This is, writing $P(\vec{x})$ as $\Sigma_i P_i^2(\vec{x})$. D. Hilbert showed that non negative polynomials can always be presented in a SOS form iff $\mathsf{degree}(P) = 2$ or $n = 1$ or $(n, \mathsf{degree}(P)) = (2, 4)$. In the general case, E. Artin showed that a non negative polynomial can always be expressed as the sum of the squares of rational functions. While a SOS form provides an efficiently checkable non negativity certificate, a negative valuation $P(\vec{x})$ may be provided to certify the contrary. Powerful semi-decision procedures for the decomposition into SOS have been implemented in HOL light [80].

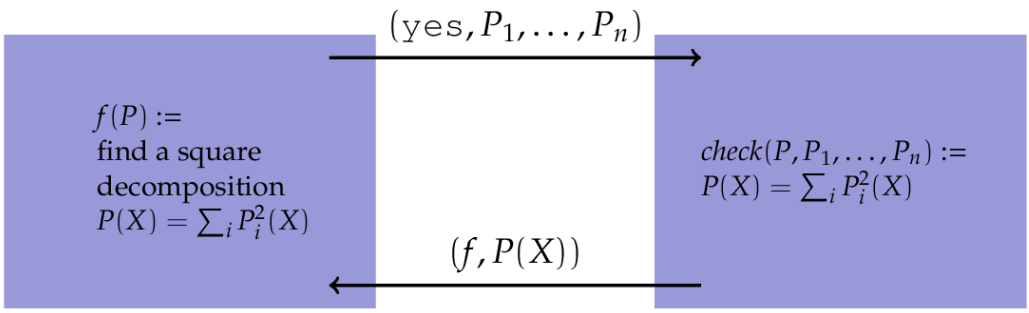
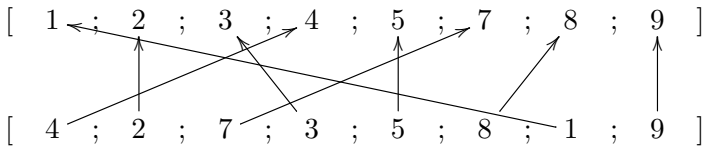


Figure 4.5: Sum of squares

- Searching for an element e in a sorted list L. Here the hint is the index i of the element ($L.i = e$) we are searching for, or the smallest number superior to it ($\forall j : 0 \leq j < i \Rightarrow L.j < e$ and $i \leq j < \#L \Rightarrow e < L.j$).

- Sorting a list L. Here the hint is the permutation P such that $P(L)$ is sorted. The permutation P may be represented as a list of pointers (indexes). In this case, a verification would consist in checking that each valid index appears exactly once. Lastly, one is left with checking $P(L)$ to be sorted which is also a task that has linear complexity with respect to the size of the original list.



- Deciding if a propositional formula is satisfiable (SAT). If the formula is satisfiable, a satisfying assignment may be provided. No succinct certificate is known for non-satisfiable formulas. However, Stålmarks algorithm provides reasonable efficiency for practical industrial applications such as railway interlocking systems. Furthermore, it allows a conceptual separation between proof searching and proof checking. This has been exploited by J. Harrison [94] to develop in his implementation of the algorithm as a HOL rule.

- Finding a maximal flow [62] in a network graph G. This problem deals with maximizing the flow f from a source node s to a sink node t respecting the edge capacities of the underlying directed graph. The max-flow min-cut theorem states that for such a flow to be maximal, there must be a cut (split of the nodes into two sets $S \ni s$ and $T \ni t$) such that all the edges from S to T are fully used by f. Standard algorithms may deliver both the flow f and the cut c without compromising the

complexity. There are many problems such as maximal matchings that may be considered refinements of the network flow problems and have themselves a refined notion of hint. Applications of these algorithm range from commodity routing to computer vision [34].

- Planarity testing is the problem of recognizing whether a graph G may be drawn in the plane without it's edges crossing. These graphs appear naturally in circuit design and many kinds of maps. If the answer is affirmative, straight-line planar-grid embeddings may be given as witness. If the graph is non-planar, Kuratowski's reduction theorem states that either the fully connected 5-vertex graph K_5 or the utility graph $K_{3,3}$ may be found as a graph minor and provided as a witness. This approach has been implemented in LEDA [115].

- Recognizing interval graphs and permutation graphs. Interval graphs are graphs in which each node represents an interval of the real numbers, and each edge represents a non empty intersection between them. They abstract relevant information from an actual set of intervals and find a number of applications in scheduling, circuit design, traffic control, genetics and other problem domains. In permutation graphs each node represents an element and edges identify the elements that change their relative ordered after a certain permutation π. Permutation graphs uniquely identifies a permutation π and allow running graph algorithms requiring a classical representation. Recognizing these types graphs are two decision problems for which efficiently checkable certificates may be provided [101]. We present this example to illustrate that result certification is not limited to the most classical problems and may be used in any context that provides an advantage to nondeterministic algorithms, either because they need to perform search or to iterate until a fixed point is found.

Certifying algorithms have not only been studied from a theoretical perspective, but they are also used in a number of practical systems, most notably for computational geometry.

The computational geometry algorithm library [42], CGAL, systematically incorporates checks into it's code. Checks can be classified into four categories: preconditions, postconditions, assertions, and warnings. In the first three cases, the code halts in the case of failure whereas the last only leads to a warning. Result checking is analogous to postcondition checking as they both verify that a routine has done what it promised to do. Both CGAL and LEDA (library of efficient data types and algorithms [115]) provide a host of domain specific functions that allow the user to verify significant assertions. These assertions are in many cases the postconditions of other routines. A repeating pattern is having an is_valid method in each class that checks the class invariant for validity.

The approach taken by both CGAL and LEDA aims at detecting implementation bugs rather than having an untrusted part perform result critical computation. The checker functions implemented in these libraries provide an algorithmic base for a PCR infrastructure. However, there are certain points that need to be addressed in a PCR infrastructure:

1. give emphasis to additional hints even if they imply more computation by the untrusted party.

2. provide a formal notion of certificates applicable in a wide range of domains.

3. describe communication channel between user and untrusted party such as *remote procedure call*.

4. formally verify result checkers for stronger correctness guarantees.

Challenge 2: infrastructure A desirable environment for producing certified results should consist of a library of certifying functions together with a systematic mechanism for combining them and producing certified programs. We distinguish two possible approaches to how to compose oracle/checker pairs to solve larger problems.

- A direct approach is to regard the different subproblems as independent (see figure 4.6). Each time the checker reaches a point where a certificate is needed, it makes a request to the corresponding

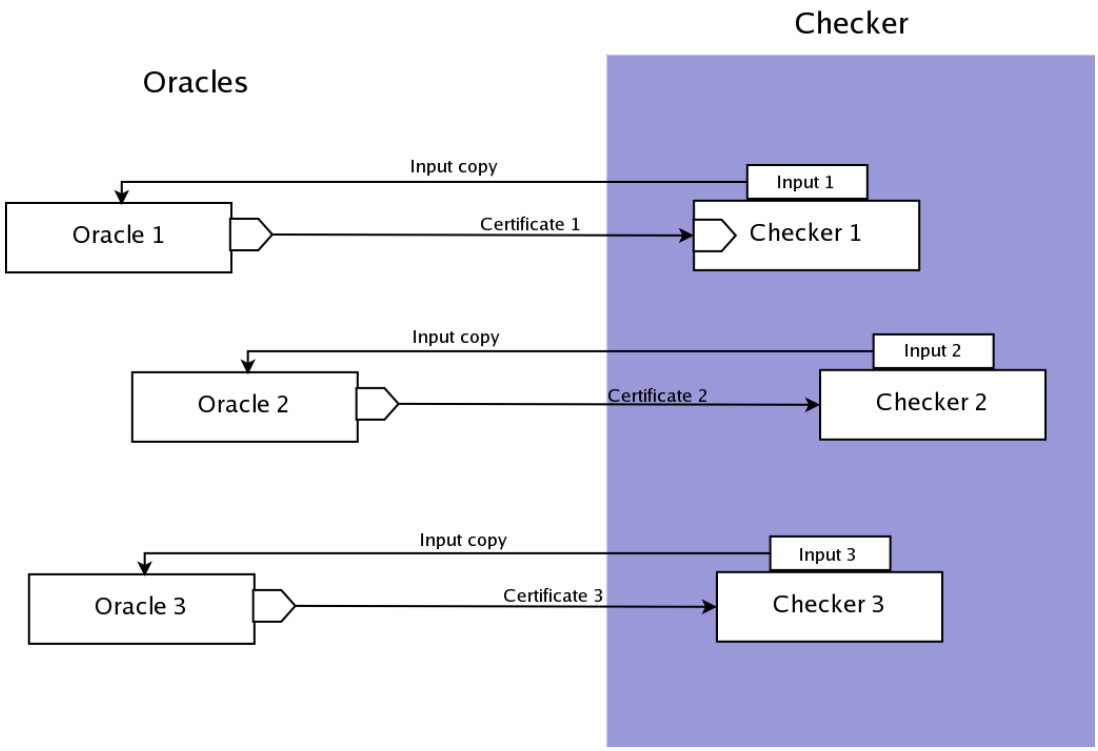


Figure 4.6: Independent oracles

oracle providing the relevant input data and subsequently receives the certified result and proceeds to check it. This approach has the advantage of allowing a straightforward distributed implementation of the oracle. Furthermore, it allows the checker to use private information in parts of the computation process. On the downside, oracles may not precalculate results, making a higher latency expectable. Furthermore, designing reduced input interfaces to oracles becomes a concern.

- Figure 4.7 represents the case in which both oracle and checker are structured in the same way. Here, the oracle and the checker execute in lock-step. This is, they maintain the same state in corresponding points of the programs and synchronize at $\text{oracle}_i/\text{checker}_i$. This requires that the $\text{oracle}_i/\text{checker}_i$ not modify the state, or modify it in exactly the same manner. In this approach, there is only one entity responsible for providing all intermediate certificates. Furthermore, the original input is consumed by both the checker and the oracle. From this point on, the oracle will be able to provide all intermediate certificates without need of further input. That is to say, the oracle knows what certificate the checker is expecting without the checker even asking for it. This allows the oracle to precalculate the certificates, even before the checker has reached the point where they are needed. Furthermore, there are many problems such as searching in a sorted list, for which an efficient checker may be defined provided there is no need for communicating the input to the oracle (i.e. passing the sorted list as parameter). A fast result checking approach is possible thanks to the identical states of oracle and the checker. However, this approach may require the oracle to execute an excessive amount of "glue code" to keep in lockstep with the checker. In addition, extra care must be taken so the glue code executes in a deterministic manner that allows the oracle to correctly predict the checker's needs.

One issue, is to develop appropriate structuring mechanisms for certificates. When checkers are not combined, hints may be structured in a static ad-hoc manner. If one wishes to compose these basic block checkers in a systematic manner to produce more complex result certifying programs, choosing a structure for communicating the hints becomes a non trivial design decision. In an imperative context, one may consider the sequential concatenation (stream) of the hints following execution order (time stamp ordering). This gives both checker and producer little or no control flow flexibility. It is thus unacceptable for functional

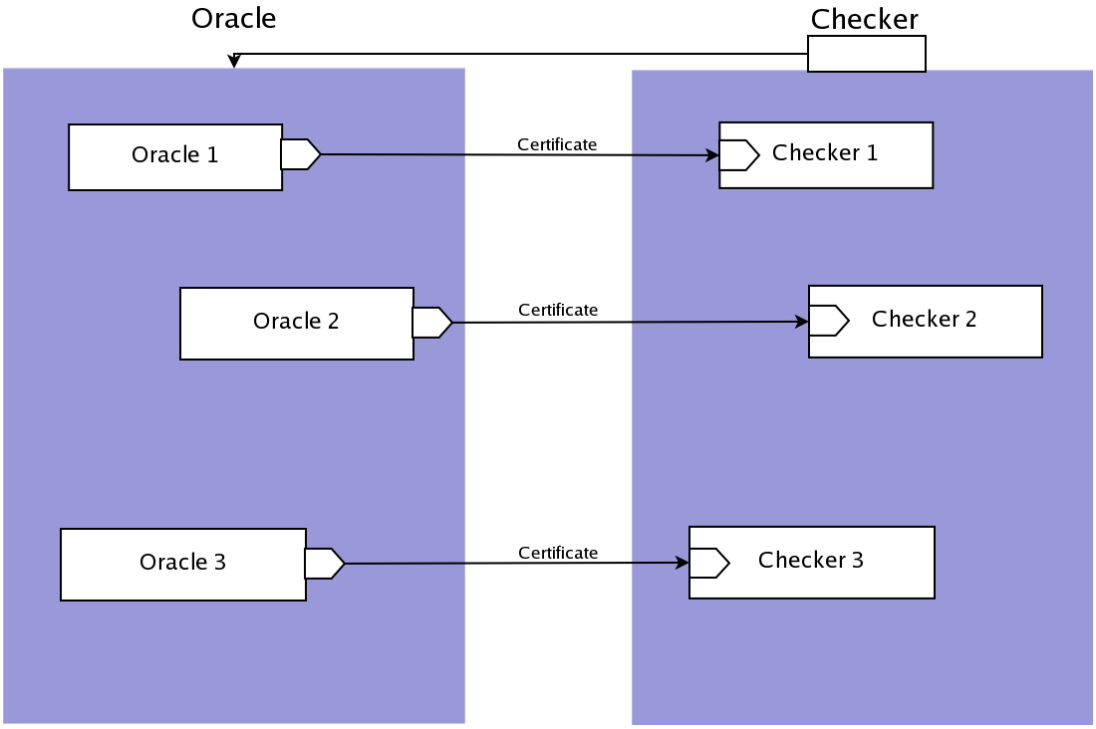


Figure 4.7: Lockstep Checker

programs, as it makes the evaluation order of subexpressions important. Another possibility, is to tag each hint with the checking context to which it corresponds (dictionary). The tags allow for flexibility in the execution order but require agreement upon a tagging method. It remains to be seen which approach can be practically implemented.

A further open question is understanding the logic required to express recursive or mutually recursive checkers. For example, in primality testing, the checker must apply the primality test on smaller numbers provided in the hint. This process continues until the numbers are so small that the checker is capable of independently verifying their primality. Since the auxiliary problems to be solved are in part determined by the untrusted part, the checker may be interested in actively enforcing termination by checking that recursive calls are made on successively decreasing arguments.

Challenge 3: interplay between PCR and program verification While the primary goal of PCR is to enable trust in distributed computations, it is also possible to exploit PCR for simplifying the task of program verification. The basic idea is to cut-off program verification tasks by isolating subroutines for which appropriate checkers exist, and by verifying these checkers instead of these subroutines. This process is illustrated in Figure 4.8, and avoids verifying complex algorithms whose correctness is not crucial to prove the correctness of the overall program (provided correct checkers can be used). For example, Xavier Leroy [107] presents a formal proof of compiler correctness in which result certification is used for checking graph colorings. In retrospect, the author advocates using result certification for verifying the intermediate results of many compiler specific algorithms such as Kildall's dataflow inequation solver, RTL type reconstruction and parallel move. One could proceed likewise for proving that certificate translation transforms valid certificates into valid certificates, provided the results of the certifying analyzer are correct. PRC hence appears as an interesting technique for program verification. In a PCC context, we can furthermore notice that, if a program incorporates result checking, then it would be easier to generate a correctness proof certificate.

Dually, a particular algorithm may be efficiently handled using PCR if several advanced certificate checkers are used simultaneously. In such cases, it is legitimate that the user questions the reliability of

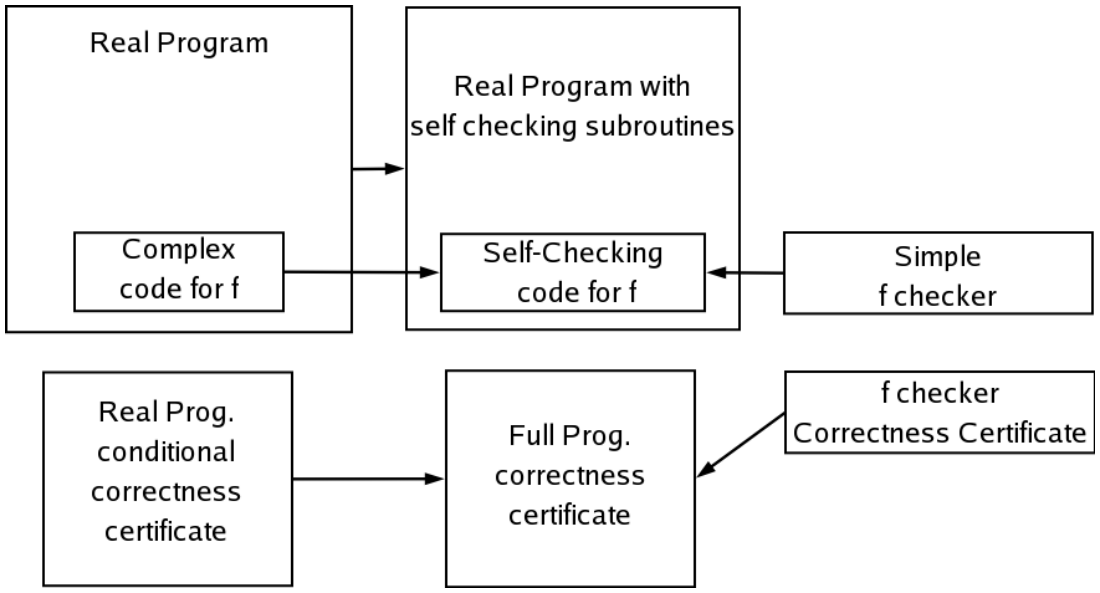


Figure 4.8: Certified program

these checkers, that are likely to originate from untrusted parties. Fortunately, PCC technology allows us to address the issue of trusting certificate checkers for PCR by requiring that certificate checkers are themselves provided with a PCC certificate, as illustrated in Figure 4.9.

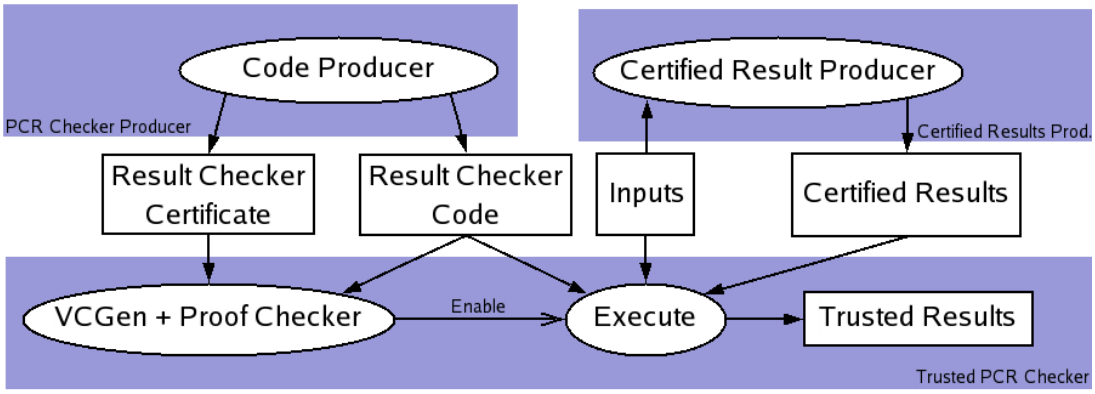


Figure 4.9: Verified Checker

4.4 Multiple Verifiers

It would be infeasible for code consumers, within global scale distributed systems, to have previous knowledge of all verifiers that might be needed during execution. Also, it could be the case that a code consumer only needs to verify a small number of properties, for example because it will execute only an isolated part of the received code. Moreover, in some configurations the code consumer may not be able to perform full proof verification due to limited computational capacity (e.g. JavaCard).

Hence, full proof verification may have to take place in a separate site, allowing verification instances to be reused, when bundled with the original code and its proof. For this scenario, code consumers still need to check compliance with their safety policies, but the code arrives pre-verified.

The basic verification protocol works as follows. A code producer generates a program together with a set of assertions (the PCC). Then the code and its proof are submitted to a trusted verifier, which performs

full verification. If the verification succeeds, the trusted verifier signs the PCC data and returns it to the code producer. The result is *pre-verified* proof carrying code.

When a code consumer receives the trusted PCC data, it submits it to the local verifier along with the local safety policy. The local verifier establishes the authenticity of the received proof, checks its correspondence with the program, and then performs verification of the local policy. The bundled (trusted) verification work is not duplicated; the original assertions are considered to hold, and are only used to try to infer the local policy. If that succeeds, the code can be executed safely at the receiver.

For example, consider a core program that accepts plugins created by different code providers. Assume that the code consumer, i.e. the core program, employs a specific security policy, for instance one constraining the communication channels that can be used by plugins, to a certain set (see [175] for a type-based approach). Using the proposed multiple verifier PCC architecture, plugins would arrive with trusted proofs of their communication properties, and the code consumer would be relieved from the computational burden of performing full proof-checking; instead, it would check whether its local policy is satisfied based on the received certificate, which in this case would mean that the local set of allowed channels should be a superset of the channels that can possibly be used in the plugin code. Notice that in order to verify this property the main application only needs to inspect the trusted certificate without having to Analise the actual plugin code.

There are manifold advantages in using this approach to verification. Firstly, the local policy of the consumer can be (or may only require) a fraction of the full assertions that were derived from the received code, and the local verifier only needs to assert that subset, with the help of the proof and the trusted verification which is reused. Secondly, this system may enable more complex and modular verification, in which several consistent logics can be used at the trusted verifiers for the sub-components of received code, with pre-verified policies composed at the local verifier to try and assert the full local policy.

The latter highlights the connection with multiple producers (Section 4.2): in a realistic scenario, mobile programs arriving at a code consumer may consist of several heterogeneous sub-components, each requiring potentially different verification techniques by specialized verifiers. Thus, multiple trusted verifiers are necessary in order to assert safety policies for multiple pieces of code composed at the consumer.

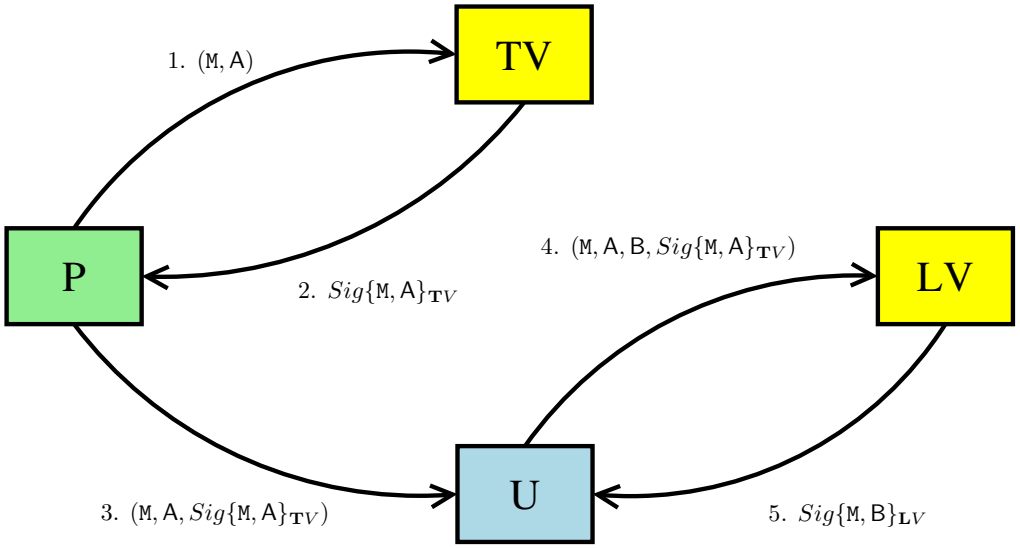
4.4.1 Security Objectives and Verification Protocol

In order for proof certificates to be trusted by code consumers, there must be some reassurance that the mentioned verifiers are genuine and indeed 'believe' that proofs hold for the received code. Thus, such verification reuse requires a trust relationship between local and remote verifiers, and specifically, that code providers and consumers agree on a set of trusted remote verifiers. The required trust relationship can be established using the trust management techniques introduced in Section 4.1. In this subsection, we explain the scenario using the notation from the standard cryptographic techniques, Public Key Infrastructure (PKI) scheme.

Our high-level security objective is to satisfy scenarios where a user **U** receives executable code M from some provider **P**. The code M arrives bundled with a set of assertions A, which have been proof-checked by a trusted verification service **TV**. Before the code M can be allowed to execute locally at **U**, it must be found to adhere to a local security policy (another set of assertions) B; this additional proof-checking task is performed by the local verification service **LV**.

Verification takes place as described by the following protocol, shown in Figure 4.10. The notation $\mathbf{P} \rightarrow \mathbf{TV} : (\mathtt{M}, \mathsf{A})$ means that **P** sends a composite message (M, A) to **TV**, which in this case constitutes a request for endorsement of the proof A of program M by the verification service at **TV**.

1. $\mathbf{P} \rightarrow \mathbf{TV} : (\mathtt{M}, \mathsf{A})$
2. $\mathbf{TV} \rightarrow \mathbf{P} : Sig\{\mathtt{M}, \mathsf{A}\}_{\mathbf{TV}}$
3. $\mathbf{P} \rightarrow \mathbf{U} : (\mathtt{M}, \mathsf{A}, Sig\{\mathtt{M}, \mathsf{A}\}_{\mathbf{TV}})$
4. $\mathbf{U} \rightarrow \mathbf{LV} : (\mathtt{M}, \mathsf{A}, \mathsf{B}, Sig\{\mathtt{M}, \mathsf{A}\}_{\mathbf{TV}})$
5. $\mathbf{LV} \rightarrow \mathbf{U} : Sig\{\mathtt{M}, \mathsf{B}\}_{\mathbf{LV}}$



TV: Trusted Verifier **LV**: Local Verifier **P**: Code Provider **U**: Code User

Figure 4.10: Verification Protocol

We write $Sig\{\mathtt{M},\mathsf{A}\}_{\mathbf{TV}}$ to denote that verifier **TV** has signed the hash values of the code M and the assertions A within a single message. After the verifier **TV** succeeds in proving the assertions, it will create such an authentic endorsement token, which the code provider can subsequently attach to the respective program. Hence, in the second message exchange the verifier **TV** returns the signed data to the provider.

Now the code provider is in a position to dispatch programs that are certified by a mutually trusted authority. This is expressed in the third message exchange, in which the code M, the respective assertions A, and the trusted verifier's token $Sig\{\mathtt{M},\mathsf{A}\}_{\mathbf{TV}}$ are sent by the code producer **P** to a potential user **U**. The user will have a local security policy B, which it may want to keep undisclosed from the code providers and remote verifiers, and before the received code can be ran it must be shown to comply with it. Since this task is to be handled by the local verifier **LV**, in the fourth message the user sends to it the code, the original assertions A, and the locally required assertions B, together with the authenticity information for the proof-checking of A. The local verifier will then verify the authenticity of the proof token, it will check that it trusts **TV** to perform proof-checking, and will finally try to infer B using A.

If the last step succeeds, the local verifier returns a new signed token, which proves to the user that **LV** indeed asserted B for M. After the protocol has been completed successfully, the code can be used at **U**.

4.4.2 Feasibility of Local Verification

In a simple setting, the proof certificate of a piece of code would be a type that can be assigned to that code, rather than a general formula: then the calculation of the entailment of the local policy verification is reduced to the decidability of the subtyping relation, reading $\mathsf{A} \geq \mathsf{B}$ (B is a subtype of A) as the entailment $\mathsf{A} \supset \mathsf{B}$ (A implies B). In the simplest case, we can envisage that the types used in the framework are the sets of communication channels, then the subtyping relation may be represented by the subset test over finite sets which is clearly decidable. The tractability of calculation may still be further retained when we augment types with simple capability such as input/output and choice information.

However, when more complex assertions are employed, the calculation of entailment in order to perform local policy checking may become increasingly difficult, and even infeasible. Hence, there exists a tradeoff between expressiveness of proofs and flexibility in defining local policies. Thus, the choice of logics to be employed, as well as its mathematical and infrastructural bases, becomes essential. We may need both a general logical framework as a foundation and its specific instantiations amenable for engineering uses such

as in local verifications. As one of the general frameworks, Task 3.1 studies a general specification logic for Java bytecode which offers a general assertion language for specifying and verifying the behavior of bytecode.

4.4.3 Process Models

In our distributed PCC scenarios, the data communicated are programs and assertions (proofs) on these programs. Naturally, in order to represent such architectures, a model of computation is required that can not only encode interactions (first-order communications) but also code mobility (higher-order interactions). Our model of choice is HOπ, the higher-order π-calculus [146], which is an integration of the π-calculus [116] and the λ-calculus. HOπ offers a rich theoretical foundation and many typing system options from the π and the λ-calculi, a number of which can be integrated and directly applicable in the systems we study here.

In our descriptions, for simplicity of presentation, we do not consider an explicit treatment of the trust architecture, and specifically of any cryptography-related operations; however, an extension using ideas from the Spi-calculus [2] and the applied π-calculus [1] is possible. Similarly, we do not consider ways to constrain which processes can communicate with each other (access control), which would be useful in modelling scenarios where, for instance, not all code providers are allowed to submit code to a specific trusted verifier; again, this would be possible if we consider fine-grained process typing extensions found in [176, 175, 81].

Process algebraic foundations

In HOπ, we write P, Q to represent processes, or agents. Agents are placed in parallel using the '|' operator, in this way modelling concurrent execution. Interaction occurs when different agents communicate a value over a channel represented abstractly as a, b, c etc. Specifically, each agent is a sequence of inputs and outputs, although it may also contain concurrency. We write $a(x:T).P$ for the process that waits to input a value of type T over channel a, binding the received value to the appropriate occurrences of x within the remaining code P; x here is treated similarly to formal arguments in procedures. The sending process is of the form $\overline{a}\langle V\rangle.Q$. When processes as the two above are composed in parallel, interaction can take place, rewriting $a(x:T).P \mid \overline{a}\langle V\rangle.Q$ to the process $P\{V/x\} \mid Q$, in which input and output over a reacted and became consumed. The notation $P\{V/x\}$ represents the process which results from substituting the (received) value V for the formal argument x in P – this substitution implements the effect of value passing interaction. In HOπ, which combines the π-calculus and the λ-calculus, the exchanged value can be a function enclosing a process, hence we have higher-order processes (processes that communicate processes).

We write $!a(x:T).P$ to express that there are infinite copies of $a(x:T).P$ in parallel; we use $(\boldsymbol{\nu}\, a)\, P$ to express that channel a is private within P; parentheses are used to disambiguate scoping of bound variables, such as input arguments. Additional notation will be explained when used.

Modelling the protocol with HOπ

The higher-order π-calculus is suitable for modelling the class of protocols involving multiple verifiers (*off-device* checking). Encoding the protocol of Figure 4.10 requires that each participant is defined as a communicating process, and that those processes are placed in parallel (i.e., run concurrently).

For example, we can model the code provider **P**, assuming channel a is a communication point between **P** and trusted verifier **TV**, and channel b between **P** and code user **U**, as follows:

$$\mathbf{P} \stackrel{\text{def}}{=} \underbrace{\overline{a}\langle \mathtt{M}, \mathsf{A}\rangle}_{output}.\overbrace{a(x:Sig)}^{input}.\underbrace{[checksig\ x\ \mathtt{M}\ \mathsf{A} == \mathtt{true}]\,(\ \overbrace{!\,b(z:\alpha).\overline{z}\langle \mathtt{M}, \mathsf{A}, x\rangle}^{iteration}\ )}_{conditional}$$

As mandated by the protocol, **P** first emits the code M and the assertions A to the trusted verifier, then waits to input the authentic response certifying the code and proof.

We assume a ternary function *checksig*, that checks whether the signature received is valid for the code and proof, returning a boolean. The expression prefixed with $[checksig\, x\, \mathtt{MA} == \mathtt{true}]$ will only ever execute if the result of the enclosed test is `true`; otherwise the computation that follows is to be skipped.

The final part of the definition (iteration), which becomes active when the signature received is valid, models that the pre-verified code is now available to clients concurrently and persistently; this is achieved by using the '!' operator that replicates the term that follows it. Clients need to send a new communication link over b, and then the provider will output the code, proof, and credentials over that new channel. In the input process, *Sig* denotes a type of a signature and α denotes a type of channels (which will be explained below).

Similar definitions can be shown for the other processes that comprise this protocol.

Types for protocol checking: session types

In the previous subsection we saw the definition, in HOπ, of the code provider **P**. Now, we juxtapose the definition of the trusted verifier **TV**, which brings us to the typing discipline of this section, whose purpose is to enable the verification of precise protocols for distributed scenarios, encoded in the HOπ language; this offers validation from a global viewpoint.

The verifier can be defined as follows:

$$\mathbf{TV} \stackrel{\text{def}}{=} \underbrace{\overbrace{a(x{:}Proc, y{:}Assert)}^{input}}_{1}.\,(\ \underbrace{(\ \lambda(z{:}Sig).\ \overbrace{\overline{a}\langle z\rangle}^{output}\ )}_{4} \cdot \underbrace{(\ createsig\_tv\ x\ y\ (\overbrace{verify\ \ x\ y}^{2})\ )}_{3}\ )$$

where the numbers are added to illustrate the order of execution that our left-to-right call-by-value operational semantics mandate. The standard λ-calculus notation $P \cdot Q$ represents function application, where P is the function and Q the operand; however, the '$\cdot$' will mainly be shown when it enhances clarity, and otherwise it may be omitted. Functions take the shape $\lambda(x{:}T).P$ where x is the formal argument of type T and P is the function body (in which x may be used). Any sound typing system ensures that the operand has the same type as the argument.

In the above definition, **TV** first waits to input a program and the assertions to be checked over a. Then, in the second step, the binary function *verify* performs the proof-checking, resulting in a boolean value. In the third step the ternary function *createsig_tv* creates a digital signature that includes information on the received program, the assertions, and the result of the *verify* application. The produced signature then becomes an operand to which the function indicated in the fourth step is applied. After that step, the digital signature will be communicated back to the code provider through the output over a.

Observing the communication patterns of **P** and **TV**, we note that they are complementary: **P** sends over a, and **TV** receives, then **TV** sends and **P** receives. Moreover, the types match throughout interaction, that is, whatever arity and type of value one process is sending, the other must be able to receive the same. Finally, the number of communications over the common channel a is equal for both terms, that is, there is not a case where one process is trying to send but there is no input possible on the other end, or vice versa. The same observations could have been made for any pair of interacting processes of the protocol.

The above are exactly the assumptions in *session types* [161], which model protocols of structured interaction over a single communication channel. Sessions have been studied in the context of π-calculus [71, 33, 166, 90], functional languages [167], remote interfaces [164], and object-oriented Java-like languages [59].

Session types are sequences of typed input and output, as defined below:

$$\alpha\ ::=\ \varepsilon\ \mid\ ![T]\ \mid\ ?[T]\ \mid\ \alpha.\alpha$$

where $![T]$ is output of type T and $?[T]$ is input of type T. The sequencing can have zero (ε) or more ($\alpha.\alpha$) actions. Note that we can easily generalise the above to allow many values to be sent/received in every step, by using a vector of types.

Sessions enable the checking of communications correspondence, and can therefore serve as a contract for the safety of protocols. The process **P** has, with respect to channel a, the session type $![Proc, Assert].?[Sig]$

which dictates that first the program (type *Proc*) and assertions (type *Assert*) will be sent (prefix !), and then that a signature (type *Sig*) will be received (prefix ?). The process **TV** has the complementary session ?[*Proc*, *Assert*].![*Sig*]. Session types faithfully capture the pairwise interactions of the protocol: we can further extend this model in order to describe and validate the original whole global scenario written in Subsection 4.4.1, based on a global description calculus with session types in [40, 90]. In these papers, not only we study a way to guarantee safety of global protocols by session types, but also we investigate type-preserving decompositions (end-point projections) from a global protocol into each local process (like **P** and **TV** above) and validate correctness of the global protocol via the translation (see [154] for an overview for its practical use). The advantage of this method in the context of our scenario is that we can use types that describe the complete verification protocol, while – through the projection to ordinary session types – retaining the tractability and decentralisation of control offered by sessions.

The first step is to investigate the integration of session types within the higher-order setting, which introduces further complexities if we are to preserve our communication invariants. The formalism being developed [119] will constitute the foundation on which other analyses will be added, like the sets-of-channels as types approach for access control of HOπ [176, 175].

4.4.4 Further Issues in Multiple-Verifier Scenario

The practical concerns of this multiple-verifier scenario, simple as it may look, raises several basic technical issues.

(1) We want the original specification from the code provider to be sufficiently detailed so that it can entail other necessary specifications for e.g. assertions for local security property. What assertion should a code provider use as the original assertion and how can it be verified?

(2) How can we combine assertions for different components coming from different sources consistently and use them for verifying an assertion of a new component, especially when complex interplay among existing components is necessary to guarantee the required security property of a new component?

The bytecode level specification language and program logic, discussed in Task 3.1, are based on Hoare logic, and can address these problems from various directions. A problem related to **(1)** is addressed in Task 3.7 (annotation generation), though the exact problem of deriving a representative assertion is not treated in that task. As a theoretical study which may suggest possible ways to address these issues, [89] discusses a method to generate most general formulae in the context of Hoare-like logics for higher-order functions [91, 92, 23].

While studied in the context of a program logic for high-level languages, the work [89] shows an automatic method for generating a full specification of a given program in an imperative call-by-value PCF. Such a specification is called *characteristic assertion*. If we can generate the characteristic assertion A of a program P, then we can check if P is to satisfy a local security property of interest, say B, by checking the implication $\mathsf{A} \supset \mathsf{B}$, instead of proving the assertion for P directly. That is, proving B for P is reduced to proving the implication $\mathsf{A} \supset \mathsf{B}$. Thus, in principle, we can use a characteristic assertion of P as a most general assertion the code provider offers, addressing the issue **(1)** above. It would be interesting to adapt the method in [89] to the bytecode logic in Task 3.1 or its variant. Since implications are not computable in general, the method in [89] may not be directly applicable. The method can however be combined with traditional methods or with provided (e.g. hand-proved) asserted programs. It also gives insight on relationship between programs and assertions.

The intention of the work in [89] is to develop a program logic for complex higher-order behaviour, which has been considered to be difficult, including combination of assertions for the multiple parts of a given program. Thus the method in [89] partly addresses the issue **(2)** above, even though its adaptation to logics for low-level code, such as the logic for bytecode studied in Task 3.1, is an open issue. The engineering potential of the method in [89] is experimented through a prototype implementation of the assertion generation algorithm, accessible from [22].

4.5 Personalization Servers

There are a number of situations in global computing where code producers will generate code which is *generic* in one way or another. In some cases, the actual code which will run on code consumers will not be such generic code but rather a *personalized instance* or *specialization* of the generic code. This scenario has some similarities with the certificate translation issue discussed in Section 3.1, but it also introduces interesting variations to the original PCC framework. A first issue is whether a certificate produced for the generic program is still *valid* for the specialized ones. We have to distinguish two main concerns here: is the generic certificate still *correct*? Is it *accurate* enough? Regarding the first question, the more general proof should still hold for the more specialized case. Therefore, there is no problem in terms of correctness. As for the second question, we might gain more accuracy by specializing the certificate for the personalized instance of the generic program. As a consequence, we might be able to entail more accurate (or strict) security policies from the specialized certificate. Therefore, it seems interesting to come up with some automatic way to personalize certificates.

The next issue is how to generate the specialized certificate. We distinguish two main approaches depending on whether a new certificate is generated for each personalization or not:

a) In the first approach, we propose to generate together with the specialization of the code a personalized certificate for it. We have developed for this purpose an *abstract interpreter with specialized definitions* [136] in which calls in the program are not analyzed w.r.t. the original definition of the procedures but w.r.t. a specialized version for them. As a result, the abstract interpreter produces a specialization of the code together with an *abstraction* for it such that, by following the ACC framework presented in Section 2.3.3, can play the role of certificate.

b) In other situations, it can appear inefficient to have to generate a different certificate for each personalization or specialization of the code. For such cases, we propose two alternative scenarios:

   b.1) One is based on the construction of *parametric certificates*. In this solution, not only the code but also the certificate is automatically instantiated for each consumer upon reception of the mobile code.

   b.2) The second solution is based on *specialization of certificates*, where we develop a mechanism for adapting a certificate to a series of transformations applied to the code.

In the above approach b.2, we can study a *taxonomy* of program transformations [126] and their implications as regards certificate validity. For example, a very simple form of specialization can consist of using just a subset of the functionality of the code. In such case, reconstructing the new certificate can be straightforward or even completely unneeded (see the discussion on deletion of procedures in Section 4.7.2). Another case which may be interesting to study is that of *program slicing* [163]. This can be considered as a simple case of program specialization where not only some parts of the code are not used, but also some arguments in method calls and some parts of method code can be deleted from a program, since they are not needed in a particular context. Here, in general, the certificate is mostly valid, but it will need to be adapted to the new program.

It is important to mention that the specialization process can be performed both at the producer and the consumer side. Both alternatives have pros and cons and seem useful in different contexts. Another idea which is interesting in principle is that of *generating extensions* [70, 30]. In our context, the mobile code can be a generating extension which produces code specialized to a particular consumer. A generating extension is a partial evaluator [96, 139] specialized with respect to a particular program. I.e., when provided with the static data it produces a specialized version. The advantage of this approach is that the consumer does not need to install a full-blown partial evaluator and that partial evaluation does not need to be done at the code producer side either.

It should be noted that code specialization can be viewed as a particular case of *incremental PCC* , as discussed in Section 4.7, in which we "update" a generic program by adding more specific information for it

and deleting the previous information about them (see Section 4.7.2). However, there might be more efficient ways for the particular case of program specialization rather than deleting and adding new procedures (see Section 4.7.2).

4.6 User Customizable Policies for On-device Checking

In existing global computing infrastructures most code producers expect their applications or components to be deployed in multiple host devices with differing capabilities and concerns about security. The actual security policy that a host device enforces on a piece of code might depend on a variety of factors including the level of trust it has in the producer and the provider, available resources, device capabilities, user configuration or the presence of other components on the device. In other cases code consumers might simply want to keep their local policy undisclosed. Therefore, in general it is not realistic to assume that every host device enforces the same security policy on a given piece of code or that policies are known a priori by the producer. As a consequence, the traditional PCC approach, where the producer gives a certificate that a piece of code abides by a security policy, cannot be directly applied because the exact policy that code must satisfy is determined by code consumers and not by code producers. Instead, an alternative PCC approach closely related to Abstraction Carrying Code (ACC) [6] can be devised. In this alternative approach, a code producer does not provide a certificate for any particular security policy, but gives instead a general abstraction of the code which allows to efficiently check host-dependent policies. As in ACC, policies are verified using abstractions of programs, but verification is performed using model-checking techniques rather than by means of static analyses. Work by Schmidt [148] shows the close relation among the two approaches.

Another PCC based approach specially targeted at the verification of resource consumption policies is that of the *Mobile Resource Guarantee* project [147] of UEDIN and LMU, which has been briefly described in Section 5.3 of Deliverable D1.1. In the MRG framework [**?**, 13], mobile code is accompanied by a verifiable certificate describing its resource behavior (*guaranteed* policy) while code consumers define their own *target* resource behavior policies. A language and a logic for specifying and reasoning about these policies are devised. By introducing further restrictions in these languages, it is possible to statically check on-device *guaranteed* policies against *target* policies. The technique described in the remainder of this section generalizes this approach by considering richer policy languages which can express general properties of programs (including resource behavior) at the cost of a more computationally intensive checking mechanism.

In the ACC framework, programs are distributed together with abstractions that play the role of certificates and are obtained by means of static analysis. The correspondence of such an abstraction with a program can be easily verified by a consumer using a simple checker that performs a single pass over the code. Using an abstraction, a consumer may then automatically generate and prove verification conditions that ensure the program satisfies a chosen security policy. In this regard, ACC already enables some degree of customization of security policies because a consumer may use the certificate (i.e. the abstraction) to verify a local policy that is not exactly the same for which the certificate was constructed, but is nevertheless implied by the given abstraction.

Although ACC is a general and practical technique, so far there exist some issues that have not been addressed. For instance, the ACC framework has been so far developed at source code level, while code suppliers in general are willing to certify and provide object code only. By building upon a trust architecture (e.g. based on PKI) it is possible to leverage the benefits of ACC and address situations where only the object code is available. The trade-off is that code consumers must trust in the abstractions provided by some verification authority instead of rechecking them. The scenario is depicted in Fig. 4.11 and explained in more detail in the paragraph below.

When given a piece of code from a producer **P**, perhaps in object form, a distributor **D** first delegates the task of generating a safe abstraction of the program to some verification authority **TV**. This task need not to be trivial, it may involve complex computations and even the use of interactive tools. Moreover, the soundness of the abstraction w.r.t. the given code may not be necessarily easily verifiable. Alternatively,

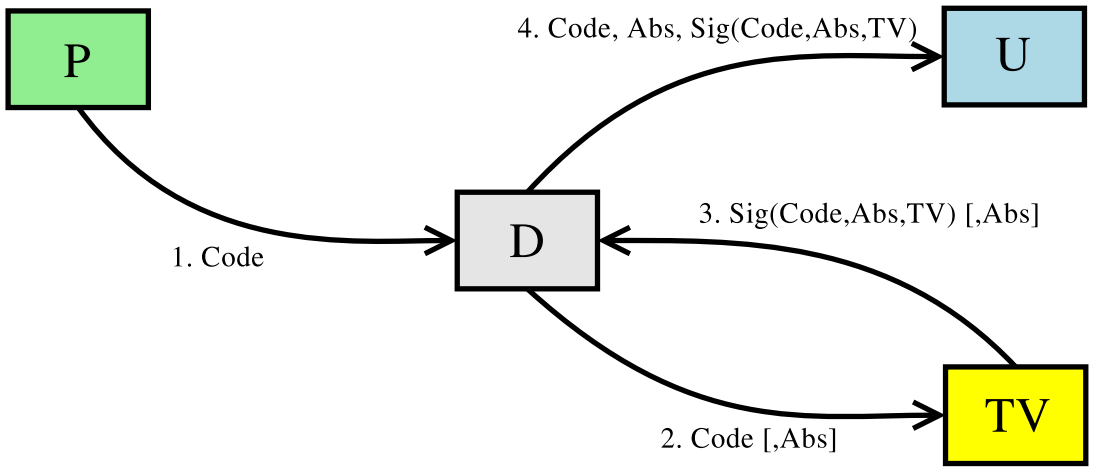


Figure 4.11: Enabling user customizable security policies by means of trust infrastructure

the distributor may directly provide an abstraction to the verification authority, whose task is in this case to verify that the abstraction is sound w.r.t. the code. The verification authority returns to the distributor the abstraction together with a signature that ties it to the program. A user **U** who obtains a copy of the program from the distributor is also sent a copy of the abstraction and the signature provided by the verification authority. The methodology generalizes naturally when multiple specialized abstractions are generated from the same program.

Once a code consumer **U** relates the origin of the abstraction to a trusted verifier, it might use it to efficiently verify a customized local security policy. This last stage is performed on-device and for most policies can be treated as an instance of model checking. We address the situation in the next subsection and propose a practical approach in the following one.

4.6.1 Towards On-device Model Checking

Model checking is a technique that consist in verifying some property of a model (i.e. an abstraction) of a system. Before any verification can begin, one is confronted with the task of choosing appropriate representations for modeling the system under study and the properties to be checked. Research on model checking in the last decades has brought a plethora of different representations for both models and properties together with verification algorithms and semi-algorithms [21, 45].

Although model checking techniques suffer from the well-known *state explosion problem* and in general require large amounts of computational resources to verify non-trivial properties of complex systems, there exist many ways to mitigate the problem. By carefully keeping the size of abstractions and properties within reasonable limits and using specialized small-footprint symbolic model checkers, checking could be performed entirely on-device. The limitations of this methodology should be determined through experimentation with realistic examples.

4.6.2 A Practical Proposal for On-device Model Checking

By means of well-established static analysis techniques it is feasible to obtain a safe model from a given program even when only object code is available. We choose to represent such models as *counter automata*, a class of automata extended with integer variables (counters) and guarded transitions whose expressions are restricted to linear integer inequalities involving at most two counters. Model checking of counter automata have been thoroughly studied in the last years [16, 49, 61, 32]. This representation is expressive enough to allow efficiently checking properties about resource consumption, access control and those that can be expressed by simpler automata classes. Of course, the proposed framework can be adapted to other

representations when other kind of policies are to be checked.

Consider the Java method `group_sms()` shown in Fig. 4.12. Using static analysis, we can derive the model represented as an automaton in Fig. 4.13, which corresponds to a call graph with respect to security sensitive APIs.

```
public void groupSMS(Phone[] group, String msg)
  throws SecurityException {

  Integer pin = getPIN();
  SecurityManager.verifyPIN(pin);

  if (group.length() > 10) {
    alert("Thou shall not spam");
    return;
  }

  for (i = 0; i < group.length(); i++)
    sendSMS(group[i], msg);

  Integer n = Read("sms.log");
  alert("You have sent " + n + " SMS so far.");
}
```

Figure 4.12: An example of a Java method that can be verified by model checking

A particular code consumer that has received from a distributor the code for the method together with its model and has verified that the model has been constructed by a trusted third party, can proceed to verify its local security policy. The security policies to be enforced may be parametrized and represented in many forms, as suggested in Table 4.1.

Table 4.1: Alternative ways to represent security policies

Description	**Representation**	**Policy**
No SMS sent before valid PIN	LTL formula	$\mathbf{G}(\neg sendSMS\ \mathbf{W}\ verifyPIN)$
No more than M credit units spent (each SMS costs N units)	Parametrized counter automaton + LTL formula	Automaton in Fig. 4.14 + $\mathbf{G}(c \leq M)$
No SMS after reading local file	Regular expression	$(\neg Read)^* + (\neg Read)^* Read(\neg sendSMS)^*$

Lets suppose that a given consumer wishes to check that the above method does not send any SMS before asking the user for a valid PIN. The policy can be represented by the following LTL formula:

$$\mathbf{G}(\neg sendSMS\ \mathbf{W}\ verifyPIN) \tag{4.5}$$

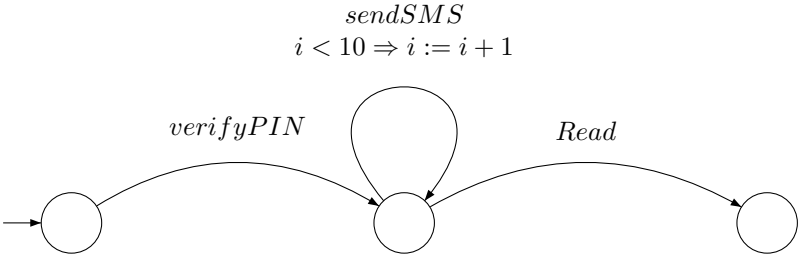


Figure 4.13: Model obtained from the Java program in Fig. 4.12

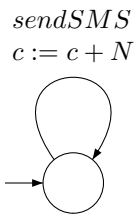


Figure 4.14: Automaton representing the behavior of an API call. Together with the LTL formula $\mathbf{G}(c \leq M)$ it determines a security policy

The consumer may also wish to verify that the method does not spend more than M credit units where the cost of each SMS is given by N and both M and N are parameters. This later property and the above LTL formula can be summarized by constructing a single counter automaton which recognizes all and only the traces of transitions labels for which at least one of the properties is violated (Fig. 4.15).

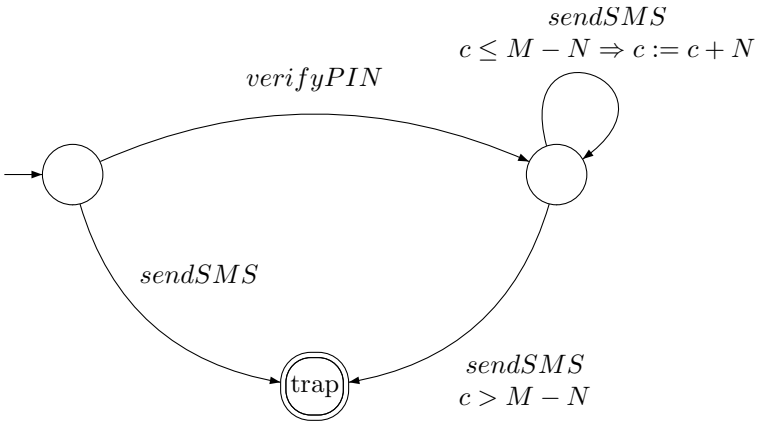


Figure 4.15: An automaton that recognizes when a composite property is violated

Before trying to verify the property, the parameters are instantiated by taking its values from the local device (e.g. user configuration, available resources). Then, the product of the automata that corresponds to the model and the property to be verified is constructed by assuming that every state in the model automaton is accepting and that transitions can only be initiated by the model (Fig. 4.16). The product automaton is also a counter automaton for which efficient reachability analysis techniques exist that may be used to model check the security policy on-device [16, 49, 61]. The outcome in this example, may depend on the particular values of the parameters M and N. Parametric model checking may be used for determining off-device the exact parameter constraints that make the method satisfy the policy (in this case $M \geq 10\,N$). Abstractions in this methodology can be regarded as a particular case of *parametric certificates* as explained in the previous section.

Another code consumer might not be interested in verifying the credit units spent by the method, but might be concerned about privacy and thus enforces a different security policy which does not allow an SMS to be sent after a local file has been read. The set of traces allowed by this policy may be described by the LTL formula $\mathbf{G}(Read \Rightarrow \mathbf{G}(\neg sendSMS))$, or by the equivalent regular expression :

$$(\neg Read)^* \; + \; (\neg Read)^* \, Read \, (\neg sendSMS)^* \tag{4.6}$$

The verification process can proceed in a similar manner using exactly the same abstraction. In this way, code consumers have the ability to compose and customize security policies without requiring code producers to be aware of them. The verification process takes place on-device and can be made in a per-policy basis or combining many policies in the same run.

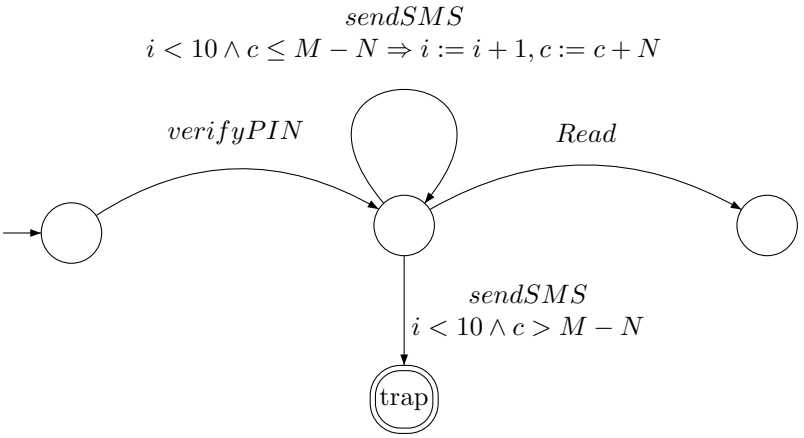


Figure 4.16: Product automaton of the automata that represent the program abstraction and the security policy

4.6.3 Analysis of Counter Automata

As said before, the problem of verifying that a program model satisfies a security property is reduced to a reachability problem on the product of the model and the property automata, which is itself a counter automaton. In general, counter automata give raise to infinite state reachability graphs. Indeed, it is well-known that even two-counter automata are as expressive as Turing machines, which means that the reachability problem is undecidable in the general case. Despite this result, it is still feasible to model check counter automata either by considering restricted subclasses for which the reachability problem is decidable, or by computing a conservative approximation of the set of reachable configurations. In either case, there exist symbolic model checking methods which represent and manipulate sets of configurations implicitly rather than by explicitly enumerating their elements. The key of these methods is the concept of *meta-transition* which allows to compute in a single step the reachable configurations by traversing a sequence of transitions from a given set of starting configurations.

Meta-transitions may accelerate the convergence of algorithms exploring the state space. However, their ultimate purpose is to enable reachability analysis algorithms to search an infinite state space, which is not possible using only simple transitions. By considering an upper-approximation of the space state, the resulting analysis is guaranteed to be safe.

4.6.4 Implementation

In order to implement the proposed methodology there are two main tasks that remain to be performed:

1. Devising static analyses techniques to synthesize program abstractions represented as counter automata;

2. Adapting existing reachability analysis algorithms of counter automata for executing them on-device.

Although these tasks are not trivial, in principle they may be completed within the next period.

4.7 Component Update and Incremental PCC

Software upgrades can be used to fix bugs or to add features to a program. Nowadays one much less often expects to install whole systems or indeed even individual applications. Instead, we are probably becoming quite used to the idea of accepting ongoing upgrades to software that we already have –a behavior of receiving incremental changes to our software systems and/or applications from vendors over the Web. Clearly, the access and deployment metaphor that the Web provides has been an important enabler for this. Given the

relatively high-speed communications infrastructure, we can expect good-sized lumps of code to come over the wire to us expediently.

Non-incremental PCC scenarios are based on checkers which receive a "certificate+program" package and read and validate the entire program w.r.t. its certificate at once, in a non-incremental way. However, situations of frequent software upgrades are not well suited to this simple model, since it would suffice with rechecking certain parts of the certificate which has already been validated. In the context of PCC, we consider possible *untrusted upgrades* of a validated (trusted) code, i.e., a code producer can (periodically) send to its consumers new upgrades of a previously submitted package. We characterize the different kind of upgrades, or modifications over a program. In particular, we include:

1. the *addition* of new data/procedures and the extension of already existing procedures with new functionalities,

2. the *deletion* of procedures or parts of them and,

3. the *replacement* of certain (parts of) procedures by new versions for them.

In such a context of software upgrades, it appears inefficient 1) to submit a full certificate (superseding the original one) and 2) to perform the checking of the entire upgraded program from scratch, as needs to be done with current systems. Herein we propose an *incremental* approach to PCC, both for the certificate generation as well as for the checking process. For the case of ACC we have already performed an initial development of incremental algorithms. More details can be found in [3, 4, **?**]. Though the particular algorithms for incremental handling of software upgrades very much depend on the particular technology used for generating certificates (ACC in the case of [**?**, 4]), it is to be expected that, similarly, algorithms for the efficient handling of incremental upgrades in type-based PCC and logic-based PCC should also be feasible.

Regarding the first issue above, i.e., when a program is upgraded, a new certificate has to be computed for the upgraded program. Such certificate differs from the original one in a) the new information for each procedure affected by the changes and b) the upgrade of certain (existing) information in the certificate affected by the propagation of the effect of a). However, certain parts of the original certificate may not have been affected by the changes. Our proposal is that, if the consumer still keeps the original certificate, it is possible to provide, along with the program upgrades, only the *difference* of both certificates, i.e., the *incremental certificate*. The first obvious advantage of the incremental approach is that the size of the certificate to be transmitted may be substantially reduced by submitting only the increment.

The second issue in incremental PCC is that the task performed by the checker can also be further reduced. In principle, a non-incremental checker requires a whole traversal of the proof where the entire program + upgrades is checked against the (full) certificate. However, it is now possible to define an *incremental checking* algorithm which, given the upgrades and its incremental certificate, only rechecks the part of the proof for the procedures which have been affected by the upgrades and, also, propagates and rechecks the effect of these changes. Thus, the second advantage of our incremental approach is that checking time is further reduced.

Efforts for coming up with incremental approaches are known in the context of program analysis (see [168, 85, 138, 145]) and program verification (see [170, 102, 152]). The ideas in this section are more closely related to incremental program analysis, although the design of an incremental checking algorithm is notably different from the design of an incremental analyzer (like the ones in [85, 138]). In particular, the treatment of deletions and arbitrary changes should be completely different. In incremental PCC, we can take advantage of the information provided in the certificate in order to avoid the need to compute the strongly connected components (see [85]). This was necessary in the analyzer in order to ensure the correctness of the incremental algorithm.

An orthogonal issue to that of efficiently checking a certificate after a software upgrade is how to make the transition from the old version of the software to the new one. In the majority of cases, this is done by halting the corresponding process and restarting it with the new upgraded code. However, there are

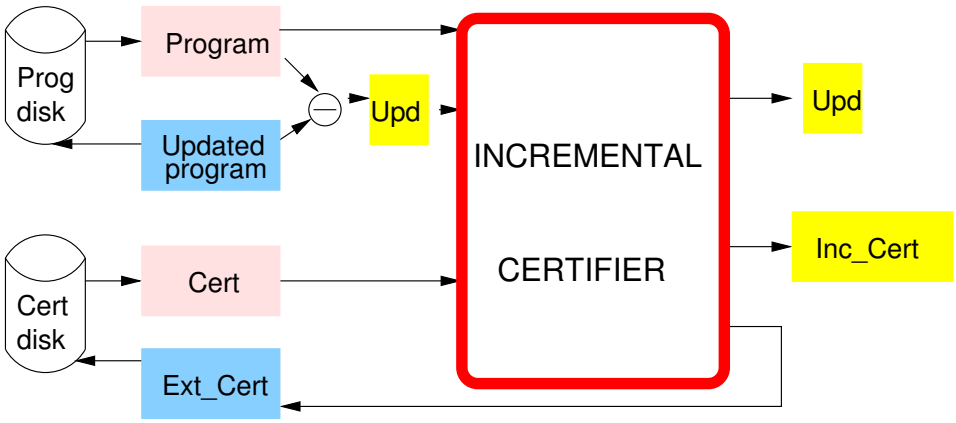


Figure 4.17: Incremental Certification

certain situations where we would like to make this transition without a *downtime*. In such cases, *dynamic* software updates (DSU) have been used for many years to fix bugs or add features to a *running* program without downtime. There is recent work [158] which provides language support to help programmers write dynamic updates effectively. In particular, Proteus was introduced as a core calculus for dynamic software updating in C-like languages. It supports dynamic updates to functions (even active ones), to named types and to data, allowing on-line evolution to match source-code evolution. The updates are guaranteed to be type-safe by checking for a property called "con-freeness" for updated types t at the point of update. This means that non-updated code will not use t concretely beyond that point (concrete usages are via explicit coercions) and thus t's representation can safely change. They also define a static updateability analysis to establish con-freeness statically, and can thus automatically infer program points at which all future (well-formed) updates will be type-safe. Later work by [120] studies the practicality of dynamic software updates (i.e., whether it is flexible, and yet safe, efficient, and easy to use). With this purpose, a DSU implementation for C that aims to meet these challenges is presented in this work. An interesting question is whether incremental PCC and the approach of [158] can be combined in such a way that DSU can take into account security properties beyond those currently considered by [158] (i.e., type safety).

4.7.1 A General View of Component Upgrade and Incremental PCC

Figures 4.17 and 4.18 provide a general view of the incremental certification process performed by the producer and the incremental checking process performed by the consumer, respectively. In Figure 4.17, the producer starts from an Updated (or upgraded) version, U_P, of a previously certified Program, P. It first retrieves from disk P and its certificate, Cert, computed in the previous certification phase. Next, the process "$\ominus$" compares both programs and returns the differences between them, $Upd(P)$, i.e, the program Upgrades which applied to P result in U_P, written as $Upd(P) = U_P \ominus P$. Note that, from an implementation perspective, a program upgrade should contain both the new upgrades to be applied to the program and instructions on where to place and remove such new code. This can be easily done by using the traditional Unix *diff* format for coding program upgrades. An Incremental Certifier generates from Cert, P and $Upd(P)$ an incremental certificate, Inc_Cert, which can be used by the consumer to validate the new upgrades. The package "$Upd(P)$+Inc_Cert" is submitted to the code consumer. Finally, in order to have a *compositional* incremental approach, the producer has to upgrade the original certificate and program with the new upgrades. Thus, the resulting Ext_Cert and U_P are stored in disk replacing Cert and P, respectively.

In Figure 4.18, the consumer receives the untrusted package. In order to validate the incoming upgrade

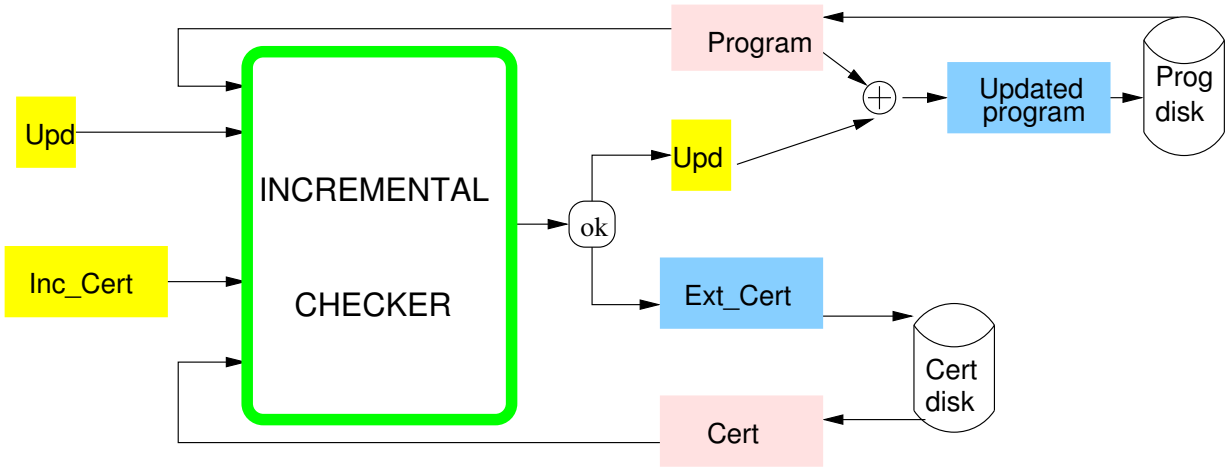


Figure 4.18: Incremental Checking

w.r.t. the provided (incremental) certificate, it first retrieves from disk P and Cert. Next, it reconstructs the upgraded program by using an operator "$\oplus$" which applies the upgrade to P and generates $U_P = P \oplus Upd(P)$. This can implemented by using a program in the spirit of the traditional Unix *patch* command as $\oplus$ operator. An Incremental Checker now efficiently validates the new modification by using the stored data and the incoming incremental certificate. If the validation succeeds (returns ok), the checker will have reconstructed the full certificate. As before, the upgraded program and extended certificate are stored in disk (superseding the previous versions) for future (incremental) upgrades. In order to simplify our scheme, we assume that the safety policy and the generation of the verification condition are embedded within the certifier and checker. However, in an incremental approach, producer and consumer could perfectly agree on a new safety policy to be implied by the modification. It should be noted that this is not covered yet by our current incremental approach, since we assume that the verification condition is generated exactly as in non incremental PCC. It is though an interesting issue for further research.

4.7.2 The Incremental Approach in the Context of ACC

In the context of ACC, we first analyze the influence of the different kinds of upgrades on the incremental approach in terms of correctness and efficiency. We also outline the main issues on the generation of incremental certificates and the design of incremental checkers.

Characterization of Upgrades

Let us first recall that in ACC, as we have seen in Section 2.3.3, the program fixpoint, $[\![P]\!]_\alpha$, automatically computed by an abstract interpretation-based analyzer, plays the role of certificate (see equations (2.1) and (2.2) in Section 2.3.3). We now characterize the types of upgrades we consider and how they can be dealt within the ACC scheme. Given a program P, we define an *upgrade* of P, written as $Upd(P)$, as a set of tuples of the form $\langle A, Add(A), Del(A)\rangle$, where $A = p(x_1, \ldots, x_n)$ is an atom in base form,[2] and:

- $Add(A)$ is the set of rules which are to be added to P for predicate p. This includes both the case of addition of new procedures, when p did not exist in P, as well as the extension of additional rules (or functionality) for p, if it existed.
- $Del(A)$ is the set of rules which are to be removed from P for predicate p.

[2]We require that each rule defining a predicate p has identical sequence of variables $x_{p_1}, \ldots x_{p_n}$ in the head atom, i.e., $p(x_{p_1}, \ldots x_{p_n})$. We call this the *base form* of p.

Note that, for the sake of simplicity, we do not include the instructions on where to place and remove such code in the formalization of our method. We distinguish three classes of upgrades: the *addition* of procedures occurs when $\forall A$, $Del(A) = \emptyset$, the *deletion* of procedures occurs if $\forall A$, $Add(A) = \emptyset$ and the remaining cases are considered *arbitrary changes*.

Addition of Procedures. When a program P is extended with new procedures or new clauses for existing procedures, the original abstraction $Cert_\alpha$ is not guaranteed to be a correct fixpoint any longer, because the contribution of the new rules can lead to a more general answer. Consider P^{add} the program after applying some additions and $Cert_\alpha^{add}$ the abstraction computed from scratch for P^{add}. Then, $Cert_\alpha \sqsubseteq Cert_\alpha^{add}$ (see Section 2.3.1). This means that $Cert_\alpha$ is no longer valid in order to entail safety. Therefore, we need to lub the contribution of the new rules and submit, together with the extension, the new certificate $Cert_\alpha^{add}$ (or the difference of both abstractions). The consumer will thus test the safety policy w.r.t. $Cert_\alpha^{add}$.

Deletion of Procedures. The first thing to note is that in order to entail the safety policy, unlike extensions over the program, we do not need to change the abstraction at all when some procedures are deleted. Consider P^{del} the program after applying some deletions and $Cert_\alpha^{del}$ the abstraction computed from scratch for P^{del}. The original certificate $Cert_\alpha$ is trivially guaranteed to be a correct fixpoint (hence a correct certificate), because the contributions of the rules were lubbed to give $Cert_\alpha$ and so it still correctly describes the contribution of each remaining rule. By applying Equation 2.2 of Section 2.3.3, $Cert_\alpha$ is still valid for P^{del} w.r.t. I_α since $Cert_\alpha \sqsubseteq I_\alpha$. Therefore, more accuracy is not needed to ensure compliance with the safety policy. This suggests that the incremental certificate can be empty and the checking process does not have to check any procedure. However, it can happen that a new, more precise safety policy is agreed by the consumer and producer. Also, this accuracy could be required in a later modification. Although $Cert_\alpha$ is a correct abstraction, it is possibly less *accurate* than $Cert_\alpha^{del}$, i.e., $Cert_\alpha^{del} \sqsubseteq Cert_\alpha$. It is therefore interesting to define the corresponding incremental algorithm for reconstructing $Cert_\alpha^{del}$ and checking the deletions and the propagation of their effects.

Arbitrary Changes. The case of arbitrary changes considers that rules can both be deleted from and added to an already validated program. In this case, the new abstraction for the modified program can be equal, more, or less precise than the original one, or even not comparable. Imagine that an arbitrary change replaces a rule R_a, which contributes to a fixpoint $Cert_\alpha^a$, with a new one R_b which contributes to a fixpoint $Cert_\alpha^b$ such that $Cert_\alpha^{ab} = \mathsf{Alub}(Cert_\alpha^a, Cert_\alpha^b)$ ($\mathsf{Alub}(CP_1, CP_2)$ performs the abstract disjunction of two descriptions) and $Cert_\alpha^a \sqsubset Cert_\alpha^{ab}$ and $Cert_\alpha^b \sqsubset Cert_\alpha^{ab}$. The point is that we cannot just abstract the new rule and add it to the previous fixpoint, i.e., we cannot use $Cert_\alpha^{ab}$ as certificate and have to provide the more accurate $Cert_\alpha^b$. The reason is that it might be possible to attest the safety policy by independently using $Cert_\alpha^a$ and $Cert_\alpha^b$ while it cannot be implied by using their lub $Cert_\alpha^{ab}$. This happens for certain safety policies which contain disjunctions, i.e., $Cert_\alpha^a \vee Cert_\alpha^b$ does not correspond to their lub $Cert_\alpha^{ab}$. Therefore, arbitrary changes require a precise recomputation of the new fixpoint and its proper checking.

Incremental Certificates

A main idea in ACC [6] is that the *certificate*, Cert, is automatically generated by using the *complete* set of entries stored in the *answer table* returned by an abstract fixpoint analysis algorithm (see Equation (2.2)). Entries in the answer table are of the form $A : CP \mapsto AP$, where A is always an atom in base form. They should be interpreted as "the answer pattern for calls to A satisfying precondition (or call pattern), CP, accomplishes postcondition (or answer pattern), AP."

If the consumer keeps the original (fixed-point) abstraction Cert, it is possible to provide only the program upgrades and the incremental certificate Inc_Cert. Concretely, given:

- an upgrade $Upd(P)$ of P,
- the certificate Cert for P and S_α,
- the certificate Ext_Cert for $P \oplus Upd(P)$ and S_α

we define Inc_Cert, the *incremental certificate* for $Upd(P)$ w.r.t. Cert, as the difference of certificates Ext_Cert and Cert, i.e., the set of entries in Ext_Cert not occurring in Cert. The first obvious advantage is that the size of the transmitted certificate can be considerably reduced.

Incremental Checkers

An abstract interpretation-based checking algorithm receives as input a program P and a certificate Cert and constructs a program analysis graph in a single iteration by assuming the fixpoint information in Cert (see Equation (2.3)). While the graph is being constructed, the obtained answers are stored in an answer table AT_{mem} and compared with the corresponding fixpoints stored in Cert. If any of the computed answers is not consistent with the certificate (i.e., it is greater than the fixpoint), the certificate is considered invalid and the program is rejected. Otherwise, Cert gets checked and $AT_{mem} \equiv$ Cert.

In order to define an incremental checking procedure, the above checking algorithm needs to be modified to compute (and store) also the dependencies between the atoms in the answer table. In [3], we have instrumented a checking algorithm for full certificates with a *Dependency Arc Table* (DAT in the following). This structure, DAT, is not required by non-incremental checkers but it is fundamental to support an incremental design. It is composed of arcs (or *dependencies*) of the form $A_k : CP \Rightarrow B_{k,i} : CP_1$ associated to a program rule A_k:-$B_{k,1}, \ldots, B_{k,n}$ with $i \in \{1, ..n\}$. The intended meaning of such a dependency is that the answer for $A_k : CP$ depends on the answer for $B_{k,i} : CP_1$, say AP_1. Thus, if AP_1 changes with the upgrade of some rule for $B_{k,i}$ then, the arc $A_k : CP \Rightarrow\ B_{k,i} : CP_1$ must be reprocessed in order to compute the new answer for $A_k : CP$. This is to say that the rule for A_k has to be processed again starting from atom $B_{k,i}$.

An incremental checking algorithm receives as input parameters an upgrade $Upd(P)$ of the original program P and the incremental certificate Inc_Cert for $Upd(P)$ w.r.t. Cert. In a very general way, the additional tasks that an incremental checking algorithm has to perform are: 1) recheck all entries in AT_{mem} which have been directly affected by an upgrade, and 2) propagate and recheck the indirect effect of these changes by inspecting the dependencies in DAT_{mem}, the dependency arc table in which the dependencies have been stored. Finally, in order to support incrementality, the final values of the data structures AT_{mem}, DAT_{mem} and P must be available after the end of the execution of the checker.

Chapter 5

Conclusions and Plans

This deliverable presents a series of scenarios which extend the original Proof Carrying Code architecture in order to cope with the situations which arise in global computing. We summarize below the different proposed scenarios. We address the scientific and technical challenges which need to be solved and whether or not the corresponding issues are to be addressed during this project. If a scenario is to be addressed, we provide information about the related work packages and particular tasks.

Instead of summarizing the scenarios in the order they have been presented in the previous chapters, we have grouped the scenarios into three categories: high-, medium-, and low-priority. Though all the proposed scenarios are of interest for global computing, it is important for the success of the project to assign priorities to the different scenarios so that those scenarios essential for the success of the project are considered first, whereas those scenarios considered to be less central to the project are only further developed if time and resources allow. More precisely, we consider *high-priority* those scenarios which we plan to study and/or deploy soon within the MOBIUS project. We classify as *medium-priority* those scenarios which we intend to study further, but later in the project. Finally, *low-priority* scenarios are those which we believe are outside of the main focus of the project and for which we cannot commit resources.

High-priority Scenarios

Enhanced Lightweight Bytecode Verification Type-based approaches to PCC (as discussed in Section 2.1), in general, and Lightweight Bytecode Verification (LBCV), in particular, enjoy certain properties which make them attractive as enabling technology for PCC, such as automation, simplicity, and efficiency, that are required for on-device checking. This is why, LBCV is the most successful and widely applied PCC technology to date. Thus, enhancing LBCV (see Section 2.1.2) to handle more advanced security properties is one of the long-standing issues for future work in PCC (see for example [106]). The aim here, as discussed in Section 2.1.3, is to keep the benefits of LBCV, including on-device checking, while addressing more advanced safety properties. This line of work is definitely within the objectives of MOBIUS and will be addressed in Task 4.5 (*On-Device Proof Checking*).

Abstraction Carrying Code Abstract Interpretation-based verification (Section 2.3) is now a well established technique with powerful static analyzers available. Abstraction Carrying Code, discussed in Section 2.3.3, carries the promise of bringing the benefits of these techniques to the context of mobile code security. While the ideas behind the initial ACC proposal [6, 6] (by members of the consortium) are language independent, they were not applied to Java bytecode. Later work [26] (also by members of the consortium) has shown that the idea of an efficient fixpoint checker on the consumer side can also be applied effectively to Java bytecode.

Though the foundations for ACC already exist, in order to make the approach useful in practice, it remains to find efficient and accurate abstract domains that are useful in proving interesting security policies about Java bytecode, including resource consumption. This is subject to ongoing and upcoming research in

Tasks 2.3, *Types for Basic Resource Policies* and 2.4, *Advanced Resource Policies*. Another interesting open question is whether on-device checking of ACC certificates is feasible in the case of resource-limited devices. For this, we need to study how to minimize both the memory and CPU usage required by the checker. This will be studied in Task 4.5.

Certificate Translation The possibility of performing certificate translation, as discussed in Section 3.1, opens up the door to bringing the benefits of source level verification to bytecode checking. It is thus a promising idea that will definitely by pursued in the project. Indeed, an important part of the effort in Task 4.4 *Proof-Transforming Compiler* will be devoted to certificate translation. Though this is a topic of ongoing work, preliminary results [17] recently obtained by members of the consortium allow for optimism about this scenario.

Certificate Formats Though the topic of certificate formats is not a "scenario" in itself, differing characteristics of certificates, such as their size and the time and space complexity of checking them, could invalidate or enable entire scenarios. As discussed in Section 3.3, there are a number of possibilities for representing certificates. First, there exist several *enabling technologies* for PCC based on different formalisms, as seen in Chapter 2. It is thus natural that certificates generated using completely different technologies contain quite different information. Furthermore, even when certificates convey the same information, different representation mechanisms may have been chosen. As it is to be expected, different formats have different trade-offs of size, complexity and efficiency of the checker, etc. Also, and as discussed in Sections 4.5 and 4.7, there are several scenarios where it is important that the certificate format has some flexibility with respect to program changes. Thus, the choice of a certificate format, especially in a complex context such as global computing, is not a trivial one and can have significant impact on the overall success of the MOBIUS technology. This issue will continue to be studied: work performed in this task is a starting point for Task 4.2 (*Certificates*).

Trusted Intermediaries The introduction of trusted intermediaries, as discussed in Section 4.1, arises from computational requirements of the PCC infrastructure, the capabilities of mobile devices, etc. It is likely that the deployment prototypes within the MOBIUS project will require the use of trusted intermediaries. Thus, the particular details of the designs of intermediaries will arise when considering such prototypes. It is important to note that *trust* is a research topic in itself and that it is outside of the scope of the project to perform detailed research in this direction. In fact we expect that, in many cases, a straightforward instantiation of standard PKI (public key infrastructure) might be enough to provide the needed level of trust required for our needs. As outlined in Section 4.1, other approaches to trust are also natural given that the key elements of the design involve logic and proof.

Multiple Verifiers It would be infeasible for code consumers, within global scale distributed systems, to have previous knowledge of all verifiers that might be needed during execution. Moreover, in some configurations the code consumer may not be able to perform full proof verification due to limited computational capacity (e.g., JavaCard). Hence, in the multiple verifiers scenario presented in Section 4.4, full proof verification may have to take place in a separate site, allowing verification instances to be reused, when bundled with the original code and its proof. Such verification reuse requires a trust relationship between local and remote verifiers. Specifically, code providers and consumers would have to agree on a set of trusted verifiers. This system can be established using standard cryptographic techniques, within security architectures such as PKI/SPKI. To guarantee the correctness of this scenario, we shall first develop a typed distributed mobile calculus which can represent the protocol and a local channel access control policy based on the work in [175, 176, 59]. Later, this approach will be augmented with a trust relationship, extending the model to a global description calculus developed in [76, 40]. These tasks will be continued within Task 4.1 itself.

Medium-Priority Scenarios

Combining Static Verification and Dynamic Checking Being able to combine static verification with dynamic checking (as discussed in Section 3.2) is interesting because it broadens the classes of security policies for which it is possible to guarantee adherence in a reasonable set of applications. Since most of interesting security policies are undecidable, having automatic tools which are both correct and complete when deciding whether an application abides by a given security policy is unfeasible. Though the state of the art in static analysis and verification steadily advances over time, and since static verification has to err on the safe side, there are always applications about which the existing tools cannot reason. Furthermore, for some complex security policies it can even be the case that no tool exists that can verify such policies statically at all.

Rather than restricting ourselves to statically verifiable security properties, dynamic checking support executing applications which are not statically guaranteed to satisfy the security policy by monitoring their execution and avoiding violations of the security policy just before they take place. A simple way of achieving this behavior is by halting those applications that attempt to violate the security policy. One important drawback of this approach is that it only makes sense for applications that are not essential to the system, as their execution is stopped. This can be acceptable for some classes of applications, but it is clearly not for others, such critical components of operating systems.

For this reason, this combination is not considered central to the project for the time being and it is not going to be the subject of much research in the next part of the MOBIUS project. It will, however, be probably be picked up again at a later point in time. In particular, we plan on integrating extended static checking and runtime assertion checking in Task 3.6 (*Program Verification Environment*).

Certified Certificate Checkers Within any PCC infrastructure, it is important to keep the trusted computing base small. Since certificate checkers used on the consumer side are part of the trusted computing base, it is important to minimize the size and complexity of such checkers. Furthermore, in the context of global computers, it is to be expected that the number of checkers that are required is not predetermined and, rather, it may become important to download new checkers which support new security policies.

As discussed in Section 3.4, in the case of abstract interpretation-based verification, it is possible to obtain certified checkers if a certified analyzer is available. This is certainly an important issue that will be addressed within the MOBIUS project. One planned approach is to generate implementations of checkers from Coq, which should increase the level of trust in certificate checker to reasonable levels. In particular, we plan to generate certified checkers for some of the resource analyses developed in Tasks 2.3 and 2.4. The feasibility of performing on-device checking using such checkers will be studied in Task 4.5.

Multiple Producers In a global computing environment it will often be the case that an application run on a code consumer device is composed of different parts originating from different producers, as discussed in Section 4.2.

An important issue in this practice is the selection of program components. In this direction, we plan on integrating multiple producer features into the MOBIUS *Program Verification Environment* in Task 3.6. In particular, consumers will be able to search for PCC components based upon the metadata contained in their generalized hybrid certificates.

The correctness of the resulting application must be based on the correctness of the separate components or modules. Thus, finding modular approaches to correctness in PCC is an important topic to MOBIUS. In fact, some work has already been devoted to modular programs in the context of ACC [127].

An important point to note here is that we are now in a situation where automated tools for reasoning about large applications are starting to be available. However, the use of such tools requires certain effort, though the long-term benefits of using such tools seem evident. Thus, there is a training issue here in order to systematically use such tools. Nevertheless, leading companies such as Microsoft are starting to impose the use of static analysis tools in software development [55, 56]. Since the topic of compositional verification is the subject of research outside the consortium, we will concentrate on the other scenarios and

reuse the existing and upcoming results on the topic of compositional reasoning within MOBIUS as such results become available.

User Customizable Policies for On-Device Checking As already mentioned, global computers are characterized by having a wide range of computing elements with greatly varying levels of computing power: some devices may, indeed, have relatively little resources. For these small devices, the checking process required by the PCC architecture will pose too high an overhead since such devices may not have the capability of performing such checking.

As a result, when designing a PCC architecture it is important to take the cost of the checking process into account. Section 4.6 presents a PCC architecture whose main feature is that the safety policies used are both user customizable and easy to check. This results in a promising approach to PCC which will be further studied in the project, in particular in Task 4.5.

Component Update and Incremental PCC As discussed in Section 4.7, one expects that the software behind global computers will be updated often. Updating may be necessary, for example, to add new functionality, to fix bugs, or to provide security patches. Clearly, it is important that certificate checking is performed as efficiently as possible, and in an incremental fashion. Thus, this scenario is of interest to MOBIUS, and will be further studied in Task 4.3 *Certificate Generation*.

Low-Priority Scenarios

Personalization Servers The possibility of having generic code which is then personalized to a given consumer, as discussed in Section 4.5, is definitely a relevant question which can be of great importance in the context of global computers, where many different devices with different features may co-exist. Though some amount of work has been devoted to this scenario during the first year of the project, further work remains in order to have mature results in this area. Thus, this issue will be further studied during the remaining 6 months of Task 4.1. It is, however, unclear whether these issues are going to be studied in other tasks outside Task 4.1.

Multiple Consumers The main scenario we have considered as regards multiple consumers is *Proof Carrying Result*, as discussed in Section 4.3.2. This is an interesting approach presented in detail in this deliverable, where a relatively comprehensive set of examples of certifying versions of some algorithms is found. However, the main drawback for its generalized deployment is the lack of an automated mechanism for finding certifying versions of an arbitrary algorithm, which is a requirement for the success of the technique.

PCC for multi-languages software and multi-logics certificates Some of the best ideas to come out of early work in component-based software engineering, as discussed in Section 4.2.2 will eventually be combined with formal specification and verification in a PCC platform like the one being built in MOBIUS.

In particular, we plan on adapting and extending the early work on structured metadata for component identification and discovery in the MOBIUS generalized certificates under development as part of Task 4.2 *Certificates*. Additionally, the formal specification matching, especially in the presence of behavioral and resources constraints [73, 100], is definitely of interest in Task 3.6 (*Program Verification Environment*). Finally, we believe that the role of evidence is critical when reasoning about the composition of multi-logic, multi-language components.

Bibliography

[1] M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *Principles of Programming Languages*, pages 104–115. ACM Press, 2001.

[2] M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997.

[3] E. Albert, P. Arenas, and G. Puebla. An incremental approach to abstraction-carrying code. Technical Report CLIP3/2006, Technical University of Madrid (UPM), School of Computer Science, UPM, March 2006.

[4] 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.

[5] E. Albert, P. Arenas, G. Puebla, and M. Hermenegildo. Reduced certificates for abstraction-carrying code. In *International Conference on Logic Programming*, number 4079 in Lecture Notes in Computer Science, pages 163–178. Springer-Verlag, August 2006.

[6] E. Albert, G. Puebla, and M. V. Hermenegildo. Abstraction-carrying code. In F. and A. Voronkov, editors, *Logic for Programming Artificial Intelligence and Reasoning*, number 3452 in Lecture Notes in Computer Science, pages 380–397. Springer-Verlag, 2005.

[7] J-M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[8] A. W. Appel. Foundational proof-carrying code. In J. Halpern, editor, *Logic in Computer Science*, page 247. IEEE Press, June 2001. Invited Talk.

[9] A. W. Appel and A. P. Felty. Lightweight lemmas in λProlog. In D. De Schreye, editor, *Joint International Conference and Symposium on Logic Programming*, pages 411–425. MIT Press, November 1999.

[10] A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Principles of Programming Languages*. ACM Press, 2000.

[11] Guillermo Arango. *Software Reusability*, chapter Domain Analysis Methods, pages 17–49. Ellis Horwood Workshop Series. Ellis Horwood, 1994.

[12] D. Aspinall. Proof general kit. White paper, version 1.6, September 2003.

[13] D. Aspinall and K. MacKenzie. Mobile resource guarantees and policies. In G. Barthe, B. Grégoire, M. Huisman, and J.-L. Lanet, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3956 of *Lecture Notes in Computer Science*, pages 16–36. Springer-Verlag, 2006.

[14] A. Avron, F. Honsell, I. A. Mason, and R. Pollack. Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9:309–354, 1992.

[15] R. Barbuti, C. Bernardeschi, and N. De Francesco. Java bytecode verification for secure information flow. *ACM SIGPLAN Notices*, 38(12):20–27, 2003.

[16] S. Bardin, A. Finkel, and J. Leroux. FASTer acceleration of counter automata in practice. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 576–590. Springer-Verlag, 2004.

[17] G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate translation for optimizing compilers. In K. Yi, editor, *Static Analysis Symposium*, number 4134 in Lecture Notes in Computer Science, pages 301–317. Springer-Verlag, 2006.

[18] G. Barthe and T. Rezk. Non-interference for a JVM-like language. In M. Fähndrich, editor, *Types in Language Design and Implementation*, pages 103–112. ACM Press, 2005.

[19] D. Basin, S. Friedrich, and M. Gawkowski. Bytecode verification by model checking. *Journal of Automated Reasoning*, 30(3-4):399–444, December 2003.

[20] Lujo Bauer, Jay Ligatti, and D. Walker. Composing security policies with polymer. In *Programming Languages Design and Implementation*, pages 305–314, New York, NY, USA, 2005. ACM Press.

[21] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, Ph. Schnoebelen, and P. McKenzie. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer-Verlag, New York, NY, USA, 1999.

[22] M. Berger. A prototype implementation of an algorithm deriving characteristic formulae. http://www.dcs.qmul.ac.uk/~martinb/capg, October 2005.

[23] M. Berger, K. Honda, and N. Yoshida. A logical analysis of aliasing for higher-order imperative functions. In *International Conference on Functional Programming*, 2005.

[24] L. Beringer, Martin Hofmann, Alberto Momigliano, and Olha Shkaravska. Automatic certification of heap consumption. In *Logic for Programming Artificial Intelligence and Reasoning*, volume 3452, pages 347–362. Springer-Verlag, 2005.

[25] A. Bernard and P. Lee. Temporal logic for proof-carrying code. In A. Voronkov, editor, *Conference on Automated Deduction*, volume 2392 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[26] F. Besson, T. Jensen, and D. Pichardie. A PCC architecture based on certified abstract interpretation. In *Emerging Applications of Abstract Interpretation*. Elsevier, 2006.

[27] T. J. Biggerstaff and A. J. Perlis, editors. *Software Reusability*, volume I: Concepts and Models of *Frontier Series*. ACM Press, 1989.

[28] T. J. Biggerstaff and A. J. Perlis, editors. *Software Reusability*, volume II: Applications and Experience of *Frontier Series*. ACM Press, 1989.

[29] T. J. Biggerstaff and C. Richter. Reusability framework, assessment, and directions. *IEEE Software*, 4(2), March 1987.

[30] L. Birkedal and M. Welinder. Hand-writing program generator generators. In *Programming Language Implementation and Logic Programming*, volume 844 of *Lecture Notes in Computer Science*, pages 198–214. Springer-Verlag, 1994.

[31] M. Blum and S. Kannan. Designing programs that check their work. *Journal of the ACM*, 42(1):269–91, January 1995.

[32] B. Boigelot. *Symbolic Methods for Exploring Infinite State Spaces.* PhD thesis, Faculté des Sciences Appliquées de l'Université de Liège, 1999.

[33] E. Bonelli, A. Compagnoni, and E. Gunter. Correspondence Assertions for Process Synchronization in Concurrent Communications. *Journal of Functional Programming*, 15(2):219–248, 2005.

[34] Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. In *Energy Minimization Methods in Computer Vision and Pattern Recognition*, pages 359–374, 2001.

[35] James M. Boyle and Monagur N. Muralidharan. Program reusability through program transformation. *IEEE Transactions on Software Engineering*, 10(5):574–588, September 1984.

[36] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the role of semantic approximations in validation and diagnosis of constraint logic programs. In *Automated Debugging*, pages 155–170, Linköping, Sweden, May 1997. U. of Linköping Press.

[37] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J.R. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. In *Workshop on Formal Methods for Industrial Critical Systems*, volume 80 of *Electronic Notes in Theoretical Computer Science*, pages 73–89. Elsevier, 2003.

[38] D. Cachera, T. Jensen, D. Pichardie, and G. Schneider. Certified Memory Usage Analysis. In J. Fitzgerald, I. Hayes, and A. Tarlecki, editors, *Formal Methods*, volume 3582 of *LNCS*, pages 91–106. Springer-Verlag, 2005.

[39] D. Cachera, D. Pichardie, T. Jensen, and G. Schneider. Certified memory usage analysis. In *Formal Methods*, number 3582 in Lecture Notes in Computer Science, pages 91–106. Springer-Verlag, 2005.

[40] M. Carbone, K. Honda, and N. Yoshida. A theoretical basis of communication-centered concurrent programming. Web Services Choreography Working Group mailing list, to appear as a WS-CDL working report.

[41] D. Caromel, L. Henrio, and B. Serpette. Context inference for static analysis of Java card object sharing. In I. Attali and T. Jensen, editors, *e-Smart*, Lecture Notes in Computer Science, pages 43–57. Springer-Verlag, 2001.

[42] Computational geometry algorithms library. http://www.cgal.org/.

[43] A. Chander, D. Espinosa, N. Islam, P. Lee, and G. C. Necula. Enforcing resource bounds via static verification of dynamic checks. In *European Symposium on Programming*, Lecture Notes in Computer Science, pages 311–325. Springer-Verlag, 2005.

[44] K. M. Chandy, B. Dimitrov, H. Le, J. Mandleson, M. Richardson, A. Rifkin, P. Sivilotti, W. Tanaka, and L. Weisman. A world-wide distributed system using Java and the internet. In *International Symposium on High Performance Distributed Computing*, 1996.

[45] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking.* MIT Press, Cambridge, Massachussets. London, England, 1999.

[46] CLDC and the K Virtual Machine (KVM). `http://java.sun.com/products/cldc`.

[47] A. Coglio. Simple verification technique for complex Java bytecode subroutines. *Concurrency and Computation: Practice and Experience*, 16(7):647–670, 2004.

[48] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. In *Programming Languages Design and Implementation*, volume 35 of *ACM Sigplan Notices*, pages 95–107. ACM Press, June 2000.

[49] H. Comon and Y. Jurski. Multiple counters automata, safety analysis and Presburger arithmetic. In A. J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 268–279. Springer-Verlag, 1998.

[50] The Concert project, Certified Code for Grid Computing. http://www.cs.cmu.edu/∼concert/.

[51] P. Cousot. Automatic verification by abstract interpretation, invited tutorial. In *Verification, Model Checking and Abstract Interpretation*, number 2575 in Lecture Notes in Computer Science, pages 20–24. Springer-Verlag, January 2003.

[52] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252, 1977.

[53] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyser. In *European Symposium on Programming*, pages 21–30. Springer-Verlag, 2005.

[54] J. Cramer, E-E. Doberkat, and M. Goedicke. Software reusability. In Wilhelm Schäfer, Rubén Prieto-Díaz, and Masao Matsumoto, editors, *Proceedings of the First International Workshop of Software Reusability*, Ellis Horwood Workshop Series, pages 79–111. Ellis Horwood, 1994. Workshop was held in July, 1991.

[55] M. Das. Formal specifications on industrial-strength code-from myth to reality. In *Computer Aided Verification*, Lecture Notes in Computer Science. Springer-Verlag, 2006.

[56] M. Das. Unleashing the power of static analysis. In *Static Analysis Symposium*, Lecture Notes in Computer Science. Springer-Verlag, 2006.

[57] M. J. Davis and H. G. Hawley. Reuse of software process and product through knowledge-based adaptation. In William B. Frakes, editor, *Proceedings of the Third International Conference on Software Reuse*, pages 44–52. IEEE Press, November 1994.

[58] P. Devanbu, R. J. Brachman, P. G. Sefridge, and B. W. Ballard. LaSSIE: A knowledge-based software information system. *Communications of the ACM*, 34(5):34–49, 1991.

[59] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session types for object-oriented languages. In *European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science. Springer-Verlag, 2006.

[60] M. Diab. Software reuse repository. In *Workshop on Software Reuse*, November 1991.

[61] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer-Verlag, 1989.

[62] P. Elias, A. Feinstein, and C. E. Shannon. Note on maximum flow through a network. *Transactions on Information Theory*, IT-2,:117–119, 1956.

[63] M. Eluard and T. Jensen. Secure object flow analysis for java card. In *Smart Card Research and Advanced Application*, pages 97–110. USENIX Association, 2002.

[64] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Programming Languages Design and Implementation*, volume 38 of *ACM SIGPLAN Notices*, pages 338–349. ACM Press, May 2003.

[65] P. Fong and R. Cameron. Proof linking: modular verification of mobile programs in the presence of lazy, dynamic linking. *ACM Transactions on Software Engineering Methodology*, 9(4):379–409, October 2000.

[66] P. W. L. Fong. Pluggable verification modules: an extensible protection mechanism for the JVM. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 404–418, 2004.

[67] R. Forno and W. Feinbloom. Inside risks: PKI: A question of trust and value. *Communications of the ACM*, 44(6):120, 2001.

[68] C. Fournet and A. D. Gordon. Stack inspection: Theory and variants. *ACM Transactions on Programming Languages and Systems*, 25(3):360–399, May 2003.

[69] S. N. Freund and J. C. Mitchell. A type system for the java bytecode language and verifier. *Journal of Automated Reasoning*, 30(3-4):271–321, December 2003.

[70] Y. Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[71] S. Gay and M. Hole. Subtyping for session types in the pi calculus. *Acta Informatica Journal*, 2005.

[72] S. Genaim and F. Spoto. Information flow analysis for Java bytecode. In R. Cousot, editor, *Verification, Model Checking and Abstract Interpretation*, volume 3385 of *Lecture Notes in Computer Science*, pages 346–362. Springer-Verlag, January 2005.

[73] J. Goguen, D. Nguyen, J. Meseguer, L., D. Zhang, and V. Berzins. Software component search. *Journal of Systems Integration*, 6:93–134, 1996.

[74] Benjamin Grégoire, Laurent Théry, and Benjamin Werner. A computational approach to pocklington certificates in type theory. *Symposium on Functional and Logic Programming*, 2006.

[75] The Infospheres Group. *The Infospheres Infrastructure version 1.0 User's Guide*. The Infospheres Group, California Institute of Technology, August 1996.

[76] Web Services Choreography Working Group. Web services choreography description language. http://www.w3.org/2002/ws/chor/.

[77] J. D. Guttman and M. Wand. Special issue on VLISP. *Lisp and Symbolic Computation*, 8(1/2), March 1995.

[78] C. Hankin, F. Nielson, and H. R. Nielson. *Principles of Program Analysis*. Springer-Verlag, 2005. Second Ed.

[79] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association of Computing Machinery*, 40(1):143–184, 1993.

[80] J. Harrison. HOL light: A tutorial introduction. In Mandayam K. Srivas and Albert J. Camilleri, editors, *Formal Methods in Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer-Verlag, 1996.

[81] M. Hennessy, J. Rathke, and N. Yoshida. safedpi: a language for controlling mobile code. *Acta Informatica Journal*, 42(4-5):227–290, 2005.

[82] L. Henrio and B. Serpette. A parameterized polyvariant bytecode verifier. In J.-C. Filliatre, editor, *Journées Francophones des Languages Applicatifs*, 2003.

[83] M. Hermenegildo, E. Albert, P. López-García, and G. Puebla. Abstraction carrying code and resource-awareness. In *Principle and Practice of Declarative Programming*. ACM Press, July 2005.

[84] M. Hermenegildo, G. Puebla, and F. Bueno. Using global analysis, partial specifications, and an extensible assertion language for program validation and debugging. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*. Springer-Verlag, 1999.

[85] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental analysis of constraint logic programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, March 2000.

[86] Manuel V. Hermenegildo, Germán Puebla, Francisco Bueno, and Pedro López-García. Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor). *Science of Computer Programming*, 58(1-2):115–140, October 2005.

[87] T. Higuchi and A. Ohori. A static type system for JVM access control. In *International Conference on Functional Programming*, pages 227–237. ACM Press, 2003.

[88] C. A. R. Hoare and J. Misra. Verified software: Theories, tools, experiments vision of a grand challenge project. A companion paper for VSTTE 2005, held in Züich, Switzerland., July 2005.

[89] K. Honda, M. Berger, and N. Yoshida. Descriptive and relative completeness of logics for higher-order functions. In *International Colloquium on Automata, Languages and Programming*, Lecture Notes in Computer Science. Springer-Verlag, July 2006.

[90] K. Honda, M. Carbone, and N. Yoshida. A calculus of global interaction based on session types. In *Workshop on Developments in Computational Models*, July 2006.

[91] K. Honda and N. Yoshida. A compositional logic for polymorphic higher-order functions. In *Principle and Practice of Declarative Programming*, pages 191–202. Association of Computing Machinery, 2004.

[92] K. Honda, N. Yoshida, and M. Berger. An observationally complete program logic for imperative higher-order functions. In *Logic in Computer Science*, pages 270–279, 2005.

[93] J. Huang and M. S. Fox. An ontology of trust: Formal semantics and transitivity. In *Conference on Electronic Commerce*, pages 259–270, New York, NY, USA, 2006. ACM Press.

[94] J. Harrison. Stalmarck's algorithm as a HOL derived rule. In J. Von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher-Order Logics*, volume 1125, pages 221–234, Turku, Finland, 1996. SV.

[95] J. W. Jo, B. M. Chang, K. Yi, and K. M. Choe. An uncaught exception analysis for Java. *Journal of systems and software*, 72(1):59–69, 2004.

[96] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[97] R. Khare and A. Rifkin. Weaving a web of trust. *World Wide Web Journal*, 2(3):77–112, 1997.

[98] G. A. Kildall. A unified approach to global program optimization. In *Principles of Programming Languages*, pages 194–206. ACM Press, 1973.

[99] J. R. Kiniry. Leveraging the world wide web for the world wide component network. In *Toward the Integration of WWW and Distributed Object Technology Workshop*, 1996.

[100] J. R. Kiniry. Semantic component composition. In *Workshop on Composition Languages*, 2003. ECOOP'03.

[101] D. Kratsch, R. M. McConnell, K. Mehlhorn, and J. Spinrad. Certifying algorithms for recognizing interval graphs and permutation graphs. In *SODA*, pages 158–167, 2003.

[102] Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 98–112, Genova, Italy, April 2001. Springer-Verlag.

[103] C. Laneve. A Type System for JVM Threads. *Theoretical Computer Science*, 290(1):741–778, October 2002.

[104] P. Lee. Certified code. Harvard University Computer Science Colloquium, January 2001.

[105] X. Leroy. On-card bytecode verification for Java card. In I. Attali and T. Jensen, editors, *e-Smart*, volume 2140 of *Lecture Notes in Computer Science*, pages 150–164. Springer-Verlag, 2001.

[106] X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3-4):235–269, December 2003.

[107] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. G. Morrisett and S. L. Peyton Jones, editors, *Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

[108] A. Levi, M. Ufuk Caglayan, and C. K. Koc. Use of nested certificates for efficient, dynamic, and trust preserving public key infrastructure. *ACM Transactions Information and Systems Security*, 7(1):21–59, 2004.

[109] N. Li, B. N. Grosof, and J. Feigenbaum. Delegation logic: A logic-based approach to distributed authorization. *ACM Transactions Information and Systems Security*, 6(1):128–171, 2003.

[110] C. Liau. A modal logic framework for multi-agent belief fusion. *ACM Transactions Computational Logic*, 6(1):124–174, 2005.

[111] Luqi. Normalized specifications for identifying reusable software. In *ACM-IEEE Computer Society 1987 Fall Joint Computer Conference*, pages 46–49, October 1987.

[112] Luqi and J. McDowell. Software reuse in specification-based prototyping. In *Workshop on Software Reuse*, November 1991.

[113] R. McDowell, D. Miller, and C. Palamidessi. Encoding transition systems in sequent calculus. *Theoretical Computer Science*, 294(3):411–437, 2003.

[114] K. Mehlhorn. The reliable algorithmic software challenge RASC. *Computer Science in Perspective*, pages 255–263, 2003.

[115] K. Mehlhorn, S. Naher, and C. Uhrig. The LEDA platform of combinatorial and geometric computing. In *Automata, Languages and Programming*, pages 7–16, 1997.

[116] Robin Milner, Joachim Parrow, and D. Walker. A calculus of mobile processes (parts I & II). *Information and Computation*, 100(1):1–77, September 1992.

[117] MOBIUS Consortium. Deliverable 3.1: Bytecode specification language and program logic, 2006. Available online from `http://mobius.inria.fr`.

[118] M. Montgomery and K. Krishna. Secure object sharing in Java Card. In *Workshop on Smart Card Technology*. Usenix, 1999.

[119] D. Mostrous and N. Yoshida. Session types and higher-order mobile processes. http://www.doc.ic.ac.uk/∼dm04.

[120] I. Neamtiu, M. W. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *Programming Languages Design and Implementation*, pages 72–83. ACM Press, 2006.

[121] G. C. Necula. Proof-carrying code. In *Principles of Programming Languages*, pages 106–119, New York, NY, USA, 1997. ACM Press.

[122] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Operating Systems Design and Implementation*, pages 229–243, Seattle, WA, October 1996. USENIX Assoc.

[123] G. C. Necula and S. P. Rahul. Oracle-based checking of untrusted software. *ACM SIGPLAN Notices*, 36(3):142–154, 2001.

[124] G. C. Necula and Robert R. Schneck. A sound framework for untrusted verification-condition generators. In *Logic in Computer Science*, page 248. IEEE Press, 2003.

[125] OpenSSL website. www. See `http://www.openssl.org/`.

[126] A. Pettorossi and M. Proietti. A comparative revisitation of some program transformation techniques. In *Dagstuhl Seminar on Partial Evaluation*, number 1110 in Lecture Notes in Computer Science. Springer-Verlag, 1996.

[127] P. Pietrzak, J. Correas, G. Puebla, and M. Hermenegildo. Context-sensitive multivariant assertion checking in modular programs. In *Logic for Programming Artificial Intelligence and Reasoning*, Lecture Notes in Computer Science. Springer-Verlag, November 2006.

[128] A. Pnueli, E. Singerman, and M. Siegel. Translation validation. In B. Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer-Verlag, 1998.

[129] J. Posegga and H. Vogt. Byte Code Verification for Java Smart Cards Based on Model Checking. In J.-J. Quisquater, Y. Deswarte, C. Meadows, and D. Gollmann, editors, *European Symposium On Research In Computer Security*, volume 1485 of *Lecture Notes in Computer Science*, pages 175–190. Springer-Verlag, 1998.

[130] F. Pottier, C. Skalka, and S. Smith. A systematic approach to static access control. *ACM Transactions on Programming Languages and Systems*, 27(2):344–382, March 2005.

[131] D. Prawitz. *Natural Deduction*. Almqvist & Wiksell, Uppsala, 1965.

[132] R. Prieto-Diaz and P. Freeman. Classifying software for reusability. *IEEE Software*, 4(1):6–16, January 1987.

[133] R. Prieto-Diaz and J. Neighbors. Modules interconnection languages. *Journal of Systems Software*, 6(4):307–334, 1986.

[134] S3MS Project. Security of software and services for mobile systems. http://www.cs.vu.nl/˜crispo/s3ms.html.

[135] G. Puebla, E. Albert, P. Arenas, and M. Hermenegildo. On abstraction-carrying code and certificate-size reduction. In *Emerging Applications of Abstract Interpretation*, March 2006.

[136] G. Puebla, E. Albert, and M. Hermenegildo. Abstract interpretation with specialized definitions. In *Static Analysis Symposium*, number 4134 in Lecture Notes in Computer Science, pages 107–126, Seoul, Korea, August 2006. Springer-Verlag.

[137] G. Puebla, F. Bueno, and M. Hermenegildo. Combined static and dynamic assertion-based debugging of constraint logic programs. In *Proceedings*, number 1817 in Lecture Notes in Computer Science, pages 273–292. Springer-Verlag, March 2000.

[138] G. Puebla and M. Hermenegildo. Optimized algorithms for the incremental analysis of logic programs. In *Static Analysis Symposium*, pages 270–284. Springer-Verlag, 1996.

[139] G. Puebla and C. Ochoa. Poly-controlled partial evaluation. In *Principle and Practice of Declarative Programming*. ACM Press, July 2006.

[140] R. Riehle. Objectivism: "Class" considered harmful. *Communications of the ACM*, 35(8), August 1992.

[141] M. Rinard. Credible compilers. Technical Report MIT/LCS/TR-776, MIT, 1999.

[142] X. Rival. Abstract interpretation-based certification of assembly code. In Lenore D. Zuck, Paul C. Attie, Agostino Cortesi, and Supratik Mukhopadhyay, editors, *Verification, Model Checking and Abstract Interpretation*, volume 2575 of *Lecture Notes in Computer Science*, pages 41–55. Springer-Verlag, 2003.

[143] E. Rose. Lightweight bytecode verification. *Journal of Automated Reasoning*, 31(3–4):303–334, 2003.

[144] M. Rosenblum and T. Garfinkel. Virtual machine monitors: Current technology and future trends. *IEEE Computer*, May 2005.

[145] B. Ryder. Incremental data-flow analysis algorithms. *ACM Transactions on Programming Languages and Systems*, 10(1):1–50, 1988.

[146] D.e Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher Order Paradigms*. PhD thesis, University of Edinburgh, 1992.

[147] D. Sannella and M. Hofmann. Mobile resource guarantees. EU Project IST-2001-33149, 2002–2005. http://groups.inf.ed.ac.uk/mrg/.

[148] D. A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Principles of Programming Languages*, pages 38–48, New York, NY, USA, 1998. ACM Press.

[149] Zhong Shao. An overview of the FLINT/ML compiler. In *ACM Workshop on Types in Compilation*, June 1997.

[150] P. Sivilotti. *A Method for the Specification, Composition, and Testing of Distributed Object Systems*. PhD thesis, California Institute of Technology, 1997. Available as Caltech technical report CS-TR-97-31.

[151] Paolo A. G. Sivilotti and Charles P. Giles. The specification of distributed objects: Liveness and locality. In *Conference of the Centre for Advanced Studies on Collaborative Research*, page 11. IBM Press, 1999.

[152] O. V. Sokolsky and S. A. Smolka. Incremental model checking in the modal **rec** -calculus. In *Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 351–363, Standford, California, USA, 1994. Springer-Verlag.

[153] M. H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Elsevier, 2006.

[154] S. Sparkes. Conversation with Steve Ross-Talbot. *ACM Queue*, 4(2), March 2006.

[155] F. Spoto, P. M. Hill, and E. Payet. Path-length analysis for object-oriented programs. In *Emerging Applications of Abstract Interpretation*, 2006.

[156] R. Stata and M. Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, January 1999.

[157] R. Steigerwald, Luqi, and J. McDowell. A CASE tool for reusable software component storage and retrieval in rapid prototyping. *Information and Software Technology*, 38(11), November 1991.

[158] G. Stoyle, M. W. Hicks, G. M. Bierman, P. Sewell, and I. Neamtiu. Mutatis mutandis: safe and predictable dynamic software updating. In *Principles of Programming Languages*, pages 183–194. ACM, 2005.

[159] M. Strecker. Formal Verification of a Java Compiler in Isabelle. In A. Voronkov, editor, *Conference on Automated Deduction*, volume 2392 of *Lecture Notes in Computer Science*, pages 63–77, London, UK, 2002. Springer-Verlag.

[160] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1997.

[161] K. Takeuchi, K. Honda, and M. Kubo. An Interaction-based Language and its Typing System. In *Parallel Architectures and Languages Europe*, volume 817 of *Lecture Notes in Computer Science*, pages 398–413. Springer-Verlag, 1994.

[162] A. Tarlecki and D. Sanella. Horizontal composability revisited. In K. Futatsugi, J. P. Jouannaud, and J. Meseguer, editors, *Algebra, Meaning and Computation. Essays dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, volume 4060 of *Lecture Notes in Computer Science*, pages 296–316. Springer-Verlag, 2006.

[163] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

[164] A. Vallecillo, V. T. Vasconcelos, and A. Ravara. Typing the behavior of objects and components using session types. In *Workshop on Foundations of Coordination Languages and Software Architectures*, volume 68(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002.

[165] R. Van Der Straeten and M. Casanova. The use of DL in component libraries - first experiences. In *Workshop on Applications of Description Logics*, Aachen, Germany, 2002.

[166] V. Vasconcelos and N. Yoshida. Language primitives and type discipline for structured communication-based programming revisited. In *Workshop on Security and Rewriting Techniques*, July 2006.

[167] Vasco T. Vasconcelos, António Ravara, and Simon Gay. Session Types for Functional Multithreading. In *Conference on Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 497–511. Springer-Verlag, 2004.

[168] T. A. Wagner and S. L. Graham. Incremental analysis of real programming languages. In *Programming Languages Design and Implementation*, pages 31–43, 1997.

[169] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.

[170] M. Wildmoser, A. Chaieb, and T. Nipkow. Bytecode analysis for proof carrying code. In F. Spoto, editor, *Bytecode Semantics, Verification, Analysis and Transformation*, volume 141 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2005.

[171] M. Wildmoser, T. Nipkow, G. Klein, and S. Nanz. Prototyping proof carrying code. In J-J. Levy, E. W. Mayr, and J. C. Mitchell, editors, *Theoretical Computer Science*, pages 333–347. Kluwer Academic Publishing, August 2004.

[172] M. Wirsing. Spectrum: A formal approach to software development with reusability. In *Workshop on Software Reuse*, November 1991.

[173] H. Xi and R. Harper. A Dependently Typed Assembly Language. In *International Conference on Functional Programming*, pages 169–180. ACM Press, 2001.

[174] F. Yang, H. Mei, Q. Wu, and B. Zhu. An approach to software development based on heterogeneous component reuse and its supporting system. *Journal of Technological Sciences*, 40(4), August 1997.

[175] N. Yoshida. Channel dependency types for higher-order mobile processes. In *Principles of Programming Languages*, pages 147–160. ACM Press, 2004.

[176] N. Yoshida and M. Hennessy. Assigning types to processes. *Information and Computation*, 172:82–120, 2002.

[177] L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic. Using replication and partitioning to build secure distributed systems. In *Symposium on Security and Privacy*, pages 236–250, Los Alamitos, CA, May 2003. IEEE Press.

[178] J. A. Zinky, D. E. Bakken, and R. E. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, January 1997.