MOBIUS: Mobility, Ubiquity, Security*

Objectives and Progress Report

Gilles Barthe[1], Lennart Beringer[2], Pierre Crégut[3], Benjamin Grégoire[1], Martin Hofmann[2], Peter Müller[4], Erik Poll[5], Germán Puebla[6], Ian Stark[7], and Eric Vétillard[8]

[1] INRIA Sophia-Antipolis, France
[2] Ludwig-Maximilians-Universität München, Germany
[3] France Télécom, France
[4] ETH Zürich, Switzerland
[5] Radboud University Nijmegen, the Netherlands
[6] Technical University of Madrid (UPM), Spain
[7] The University of Edinburgh, Scotland
[8] Trusted Labs, France

Abstract. Through their global, uniform provision of services and their distributed nature, global computers have the potential to profoundly enhance our daily life. However, they will not realize their full potential, unless the necessary levels of trust and security can be guaranteed.

The goal of the MOBIUS project is to develop a Proof Carrying Code architecture to secure global computers that consist of Java-enabled mobile devices. In this progress report, we detail its objectives and provide a snapshot of the project results during its first year of activity.

1 Introduction

Global computers are distributed computational infrastructures that aim at providing services globally and uniformly; examples include the Internet, banking networks, telephone networks, digital video infrastructures, peer-to-peer and ad hoc networks, virtual private networks, home area networks, and personal area networks. While global computers may deeply affect our quality of life, security is paramount for them to become pervasive infrastructures in our society, as envisioned in ambient intelligence. Indeed, numerous application domains, including e-government and e-health, involve sensitive data that must be protected from unauthorized parties. Malicious attackers spreading over the network and widely disconnecting or disrupting devices could have devastating economic and social consequences and would deeply affect end-users' confidence in e-society. In spite of clear risks, provisions to enforce security in global computers remain extremely primitive. Some global computers, for instance in the automotive industry, choose to enforce security by maintaining devices completely under the control

* Work partially supported by the Integrated Project MOBIUS, within the Global Computing II initiative.

U. Montanari, D. Sannella, and R. Bruni (Eds.): TGC 2006, LNCS 4661, pp. 10–29, 2007.

of the operator. Other models, building on the Java security architecture, choose to enforce security via a sandbox model that distinguishes between a fixed trusted computing base and untrusted applications. Unfortunately, these approaches are too restrictive to be serious options for the design of secure global computers. In fact, any security architecture for global computing must meet requirements that reach beyond the limits of currently deployed models.

The objective of the MOBIUS project is to develop the technology for establishing trust and security in global computers, using the Proof Carrying Code (PCC) paradigm [37,36]. The essential features of the MOBIUS security architecture are:

- *innovative trust management*, dispensing with centralized trust entities, and allowing individual components to gain trust by providing verifiable certificates of their innocuousness; and
- *static enforcement mechanisms*, sufficiently *flexible* to cover the wide range of security concerns arising in global computing, and sufficiently *resource-aware* and *configurable* to be applicable to the wide range of devices in global computers; and
- *support for system component downloading*, for compatibility with the view of a global computer as an evolving network of autonomous, heterogeneous and *extensible* devices.

MOBIUS targets are embedded execution frameworks that can run third party applications which must be checked against a platform security policy. In order to maximize its chances of success, the MOBIUS project focuses on global computers that consist of Java-enabled devices, and in particular on devices that support the Mobile Information Device Profile (MIDP, version 2) of the Connected Limited Device Configuration (CLDC) of the Java 2 Micro Edition.

2 MIDP

CLDC is a variant of Java for the embedded industry, and stands between JavaCard and Java Standard Edition. CLDC is a perfect setting for MOBIUS because it has all the characteristics of a real language: true memory management, object orientation, etc., but applications developed for it are still closed: there is no reflection API, no C interface (JNI) and no dynamic class loading (class loading is done at launch time). Furthermore, CLDC is widely accepted by the industry as a runtime environment for downloadable code: on mobile phones (MIDP), set-top-boxes (JSR 242) and smart card terminal equipment (STIP).

The MIDP profile is a set of libraries for the CLDC platform that provides a standardized environment for Java applications on mobile phones (so-called midlets). Its wide deployment (1.2 billion handsets) has lead to a consensus on security objectives. Moreover, MIDP promotes the idea of small generic mobile devices downloading services from the network and is an archetypal example of the global computing paradigm.

MIDP defines a simple connection framework for establishing communications over various technologies, with a single method to open a connection that takes as argument a URL which encodes the protocol, the target address, and some of the connection

parameters. MIDP offers a graphical user interface implementing the view/controller paradigm and provides access to specific mobile phones resources (persistent store, players, camera, geolocalisation, etc.).

MIDP security policy is based on the approval by the end-user of every method call that can threaten the security of the user (such as opening a network connection). Depending on the API, the frequency of security screens varies (from once for all to once for every call).

This scheme, although simple, has several drawbacks: users accept dangerous calls one at a time and have no idea of the forthcoming calls necessary for the transaction; there can be too many screens to perform a simple transaction; moreover even a clearly malicious action will be statistically accepted by some users if the customer basis is large enough. To mitigate some of these risks, MIDP2.0 proposes to sign midlets. Signing changes the level of trust of the midlet and reduces the number of mandatory warning screens. Signing moves the decision of accepting an API call from the end-user to a trusted entity (the manufacturer, the operator or an entity endorsed by them), but it does not provide clues to take the decision. One goal of MOBIUS is to develop the necessary technology for allowing the developer to supply clues and proofs that can help operators to validate midlets developed by third parties.

Finally, MIDP dynamic security policy does not provide any control on the information flow. This is in contrast with the european legislation that puts information control at the heart of its requirements for computerized systems [38]. The information flow analysis reported in Section 5.3 provides a first step to provide a technical enforcement of those regulations.

Several factors such as handset bugs, different handset capabilities, operational environment (language, network), etc. lead to a fragmentation of MIDP implementations. As resources (cpu, memory for heap, code or persistent data) on device are scarce, code specialization is the only viable alternative to adapt application to handsets. It is not uncommon to have hundreds of versions of a single application. Whereas some solutions exist for automating the development, the management, and the provisioning to the handset of so many variants, in practice, validation [32] is still based on a technology which is unable to cope with multiple versions: black-box testing. Indeed, only the bytecode is available to test houses, as software companies refuse to disclose their source code to third parties to protect their intellectual property. MOBIUS outcome should help to automate the validation process for operators. PCC can be used on the most complex properties whereas type based techniques could be sufficient on simple ones.

3 PCC Scenarios

Figure 1 shows the basic structure of all certificate-based mobile code security models, including Proof Carrying Code. This basic model, or *scenario*, comprises a code *producer* and a code *consumer*. The basic idea in PCC is that the *code* is accompanied by a *certificate*. The certificate can be automatically and efficiently checked by the consumer and it provides verifiable evidence that the code abides by a given *security policy*. The main difference w.r.t. digital signatures is that the latter allows having certainty on the *origin* of the code, whereas PCC allows having certainty about the

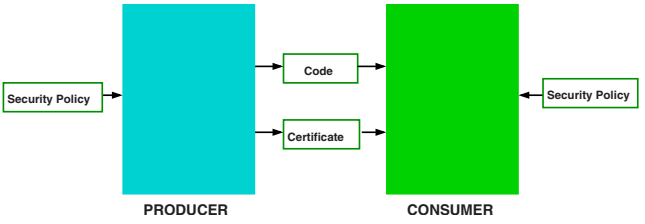

Fig. 1. Certificate-based Mobile Code Security

behaviour of the code. Different flavours of PCC exist which use different techniques for generating certificates, ranging from traditional logic-based verification to static analysis in general and type systems in particular.

In the context of global computing, this initial scenario needs to be extended in a number of ways to consider the presence of multiple producers, multiple consumers, multiple verifiers and intermediaries. We have identified a series of innovative scenarios for applying Proof Carrying Code in the context of global computers [23]; below we summarize the main scenarios and issues of interest within the context of MOBIUS.

3.1 Wholesale PCC for MIDP Devices

Figure 2 depicts the MOBIUS scenario for MIDP devices. It involves a trusted intermediary (typically the mobile phone operator), code producers that are external to the phone companies, and code consumers (the end users). PCC is used by developers to supply phone operators with proofs which establish that the application is secure. The operator then digitally signs the code before distributing it to the user.

This scenario for "wholesale" verification by a code distributor effectively combines the best of both PCC and trust, and brings important benefits to all participating actors. For the end user in particular, the scenario does not add PCC infrastructure complexity to the device, but still allows effective enforcement of advanced security policies.

From the point of view of phone operators, the proposed scenario enables achieving the required level of confidence in MIDP applications developed by third parties through formal verification. Although this process is very costly, which often results in third party code not being distributed, PCC enables operators to reproduce the program verification process performed by producers, but completely automatically and at a small fraction of the cost.

From the software producer perspective, the scenario removes the bottleneck of the manual approval/rejection of code by the operator. This results in a significant increase in market opportunity. Of course, this comes at a cost: producers have to verify their code and generate a certificate before shipping it to the operator, in return for access to a market with a large potential and which has remained rather closed to independent software companies.

3.2 Retail PCC and On-Device Checking

Although our main MOBIUS scenario is for wholesale proof-checking by a trusted intermediary, we are also exploring possibilities for "retail" PCC where checking

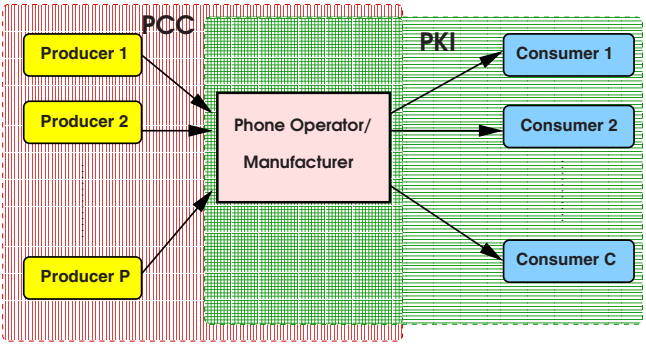

Fig. 2. The MOBIUS scenario

takes place on the device itself. Limited computing capabilities rule out full-blown proof-checking for the moment, but there are other kinds of certificates that support verification: MIDP already annotates code with basic type information for *lightweight bytecode verification* [40], and we aim to extend this with more sophisticated types to capture security properties, and with the results of other analyses as in *abstraction-carrying code* [1]. Complementary to digital signatures, these certificates maintain the PCC property that clients perform actual verification of received code, by providing rich type information to make it fast and cheap to do.

3.3 Beyond the MOBIUS Scenarios

Though the MOBIUS scenario concerns networks of mobile devices, we believe that the concept of trusted intermediary and the use of off-device PCC can have a significant impact in the quality of the applications developed in other contexts. For the case of general-purpose computers, we believe that our scenario is also applicable, since the role of trusted intermediary can be played by other organizations such as end-user organizations, governmental institutions, non-profit organizations, private companies, etc. Note that this scenario is radically different from the situation today: though some organizations play the role of trusted intermediaries, they do not have the technology for formally verifying code and they have to resort to other techniques such as manual code inspection. Thus, we argue that PCC holds the promise of bringing the benefits of software verification to everyone. The fact that verified code becomes available at low cost will increase the demand on verified code, which will in turn encourage software companies to produce verified code with certificates.

4 Security Requirements

A fundamental question in developing a security architecture for global computers is the inventory of the security requirements that we should be able to express and guarantee. This has been the one of the first step of the project.

The choice to focus on the MIDP framework was very helpful, as it allowed us to consider concrete examples of various kinds of security requirements. Moreover, as the framework has been actively used for some time, there is considerable experience with security requirements for MIDP applications. Although inspired by concrete MIDP setting, or even concrete MIDP ahpplications, the range of security requirements we have found is representative of the requirements that are important for any distributed computing infrastructure.

We have considered two, largely orthogonal ways to analyse and classify security requirements. In a first deliverable [19], we investigated two important classes of security requirements, namely *resource usage* and *information flow*. In a second one [20] we considered general security requirements that apply to all applications for the MIDP framework, so-called *framework-specific* security requirements, and security requirements specific to a given application, so-called *application-specific* security requirements. Here we summarise the main conclusions of those reports.

4.1 Resources

Any global computing infrastructure naturally raises issues about identifying and managing the resources required by mobile code. This is especially true on small devices, where resources are limited.

Central issues for resource policies are: what resources they should describe; how resource policies can contribute to security; and what kinds of formalism are appropriate. Surveying different possible kinds of "resource", we are looking to identify those that are both likely to be amenable to formal analysis by current technologies, and are also clearly useful to real-world MIDP applications. Some of these are classical instances of computational resources, namely *time*, where counting bytecodes executed can be a useful estimate of actual runtime, and *space*, of stack or heap, which may be rather limited on a mobile device. The focus on MIDP also allows us to address some platform-specific kinds of resource, namely *persistent store*, as file storage space will be limited, and *billable events* such as text messages (SMS) or network connections (HTTP), which have real-money costs for the user. Many of these platform-specific resources can be unified by treating particular system calls as the resource to be managed: how many times they are invoked, and with what arguments. This fits neatly into the existing MIDP security model, where certain APIs are only available to trusted applications.

Policies to control resources such as these are useful in themselves, but they also have a particular impact on security. First, some platform-specific resources are intrinsically valuable — for example, because an operator will charge money for them — and so we want to guard against their loss. Further, overuse of limited resources on the device itself may compromise *availability*, leading to denial of service vulnerabilities.

4.2 Information Flow

Information policies can track integrity or confidentiality. We concentrated on the second, as the former is essentially just its dual. The attacker model is a developer who leaks sensitive information to untrusted parties, either intentionally (in case of a malicious developer) or by accident. On the MIDP platform sensitive information

is typically information related to the user: sources include the addressbook, audio or video capture, the permanent store, and textfields where the user typed in private data. Untrusted information sinks are network connections and the permanent store, especially if the store is shared between applications.

4.3 Framework-Specific Security Requirements

Framework-specific security requirements describe generic requirements applicable to all the applications running on a given framework. In industry there is already considerable experience with framework-specific security requirements for MIDP. [20] provides a comprehensive listing of all of these requirements.

Many of these requirements concern critical API methods: both the use of certain methods (does the application uses the network ?) and possibly also the arguments supplied to them (for example the URL supplied to open a connection defines the protocol used). Deciding these questions is already an issue in the current MIDP code-signing scheme: to decide if signing is safe, it is necessary to know statically which critical APIs are used and to compute an approximation of the possible values of their key parameters. There are already some dedicated static analysis techniques for this [16,24], but there is a limit to what such automated analyses can achieve.

More complicated requirements on API methods are temporal properties that involve the sequencing of actions, such as a requirement that every file that is opened must be closed before the program exits. Checking these properties requires a deeper insight of the control flow of a program, which can be complicated by the possibility of runtime exceptions, the dependency on dynamic data structures, and the influence of thread synchronization. Finite state automata are a convenient formalism for specifying temporal requirements. Such automata can be expressed in the program specification language JML that we plan to use. Moreover, they are easily understandable by non-experts.[1]

4.4 Application-Specific Security Requirements

An individual application may have specific security requirements beyond the generic requirements that apply to all the applications. These application-specific security requirements may simply be more specific instances of framework-specific security properties, but can also be radically different. Whereas framework-specific requirements are often about the absence of unwanted behaviour, security requirements for a particular application may include functional requirements, concerning the correctness of some functional behaviour. Application-specific properties are usually more complex than framework-specific properties and less likely to be certified by fully automatic techniques.

We have selected some archetypical applications representative of classical application domains for which interesting security requirements can be expressed. These applications include a secure private storage provider, an instant messenger client, an SSH

[1] In fact, the current industrial standard for testing MIDP applications, the Unified Testing Criteria [32] already uses finite automata for specification, albeit informally.

client, and an application for remote electronic voting. All of these have strong security requirements, including information flow requirements, that go beyond the framework-specific requirements.

The final two applications selected are in fact core services of the MIDP platform itself rather than applications that run on the platform, namely a bytecode verifier and a modified access controller. Note that for these components functional correctness is one of the security requirements. The specification language JML that we will use in logic-based verification is capable of expressing such functional requirements, although extensions to conveniently use mathematical structures in specification, as proposed in [15], may be needed to make this practical.

5 Enabling Technologies

A central component of the technology being developed by MOBIUS is a hierarchy of mechanisms that allow one to reason about intensional and extensional properties of MIDP-compliant programs executed on a Java Virtual Machine. The two enabling technologies that these mechanisms rely on are *typing* and *logic-based verification*. Depending on the security property, and the respective computational resources, code producer and consumer (or verifier in the case of wholesale PCC) may negotiate about the level at which the certificate is formulated. For example, the availability of a type system with an automated inference algorithm reduces the amount of code annotations, whereas expressive program logics may be applied in cases when type systems are insufficiently flexible, or when no static analysis is known that ensures the property of interest. In the sequel, we provide a short overview of the mechanisms developed during the first year of the project, namely the MOBIUS program logic for sequential bytecode, and type systems for resources, information flow, and aliasing.

In the following sections we summarise some of the formal systems which we have developed and outline possible verification approaches.

5.1 Operational Model

The lowest level of our hierarchy of formal systems consists of an operational model of the Java Virtual Machine that is appropriate for MOBIUS. In particular, as a consequence of the choice to target the MIDP prifile of the CLDC platform, features such as reflection and dynamic class loading may safely be ignored, as is the case for complex data types. In addition, our current model is restricted to the sequential fragment of the JVM and does not model garbage collection.

The operational model builds the basis for all program verification formalisms to be developed in MOBIUS: all formal systems considered within the MOBIUS project – and hence the validity of certificates – may in principle be given interpretations that only refer to the operational judgments defining the model. Like any mathematical proof, these interpretations may involve some abstractions and definitional layers, including some more abstract operational semantics which we have defined and formally proven compatible with the small-step relation.

In order to guarantee the utmost adherence to the official specification, we have implemented a small step semantics. The corresponding judgement relates two consecutive states during program execution. We keep the same level of detail as the official description, but with some simplifications due to the fact that we concentrate on the CLDC platform.

The correctness of an operational model can not be formally proved, we assert it axiomatically, and have developed a rigorous mathematical description of it, called Bicolano, in the Coq proof assistant [43]. In order to get more confidence in our axiomatization we have also developed an executable version of fragments of Bicolano which can be used to compare evaluation results with other implementations of the official specification.

5.2 Program Logic

The second layer of our reasoning infrastructure is built by a program logic. This allows proof patterns typically arising during the verification of recursive program structures to be treated in a uniform matter. Extending program logics with partial-correctness interpretations, the MOBIUS logic supports the verification of non-terminating program executions by incorporating strong invariants [28].

The global specification structure is given by a table M that associates a partial-correctness method specification Φ and a method invariant φ to each defined method, where the latter relates each state occurring throughout the (finite or infinite) execution of the method to its initial state. In order to support the modular verification of virtual methods, the method specification table is required to satisfy a *behavioural subtyping* condition which mandates that the specification of an overriding method declaration must be stronger (i.e. imply) the specification of the overwritten method. In addition, each program point in a method may be decorated with an assertion that is to be satisfied whenever the control flow passes through the decorated program point. All such annotations are collected in a global annotation table Q.

The program logic employs proof judgements of the form $G \vdash \{A\}\, \ell\, \{B\}\ (I)$ where the program point ℓ (comprising a method identifier M and a label in the definition of M's body) is associated with a (local) precondition A, a local postcondition B, a (strong) invariant I. The types and intended meanings of these components are as follows.

Whenever the execution of M, starting at label 0 and initial state s_0 reaches ℓ with current state s, and $A(s_0, s)$ holds, then

- $B(s_0, s, t)$ holds, provided that the method terminates with final state t,
- $I(s_0, s, H)$ holds, provided that H is the heap component of any state arising during the continuation of the current method invocation, including invocations of further methods, i.e. subframes,
- $Q(s_0, s')$ holds, provided that s' is reached at some label ℓ' during the continuation of the current method invocation, but not including subframes, where $\mathsf{Q}(\ell') = Q$.

Moreover, the judgements are supplied with a proof context G. The latter contains assumptions typically associated with merge-points in the control flow graph. These assumptions are used by the logic rules in order to avoid infinite cycling in the proof derivation. For the technical details of this the reader is referred to [22,9].

In order to give a flavor of what the proof rules look like, we show the rule for basic instructions (arithmetic operations, load/store,...):

$$\text{INSTR}\,\frac{G \vdash \{Pre_{M,l}(A)\}\, M, suc_M(l)\, \{Post_{M,l}(B)\}\, (Inv_{M,l}(I)) \qquad \psi}{G \vdash \{A\}\, M, l\, \{B\}\, (I)}$$

Note that the correctness of l depends on the correctness of its successor. Also, the rule uses predicate transformers $Pre_{M,l}(A)$,$Post_{M,l}(A)$, and $Inv_{M,l}(I)$ which relate the assertions for the successor instruction with the assertions of instruction l. For the definition of these transformers, see [9]. Finally, the side condition ψ states that the local precondition A implies the strong invariant I and any annotation that may be associated with M, l in the annotation table Q:

$$\psi = \forall\, s_0\ s.\ A(s_0, s) \rightarrow (I(s_0, s, heap(s)) \wedge \forall\, Q.\ \mathsf{Q}(M, l) = Q \rightarrow Q(s_0, s)).$$

In addition to rules of similar shape for all instruction forms, the logic is also supplied with logical rules, such as a consequence rule and an axiom rule that extracts assumptions from the proof context.

We have proven a soundness theorem for the proof system which ensures that the derivability of a judgement $G \vdash \{A\}\, \ell\, \{B\}\, (I)$ entails its semantic validity. The latter is obtained by formulating the above informal interpretation in terms of Bicolano's operational judgements.

This soundness result may subsequently be extended to programs. We first say that a program has been *verified* if each entry in the method specification table is justified by a derivation for the corresponding method body, and similarly for the entries of local proof contexts G. The soundness result for programs then asserts that all methods of a verified program satisfy their specifications: whenever $\mathsf{M}(M) = (\Phi, \varphi)$ holds, any invocation of M is guaranteed to fulfill the method invariant φ, with terminating invocations additionally satisfying the partial-correctness assertion Φ.

In order to evaluate our logic experimentally, we have implemented a verification condition generator (VCgen) that applies proof rules in an automatic fashion and emits verifications conditions stemming from side conditions such as ψ above, and from the application of the rule of consequence.

In the next period of the project, we will extend the logic by mechanisms for reasoning about the consumption of resources and incorporate ghost variables and associated concepts. This will provide a platform for the encoding of some type systems that defy the current version of the program logic. A typical example are type systems that track the number of calls to certain API-methods like sending of SMS messages or opening files.

5.3 Type Systems

In this section we describe MOBIUS work on types for information flow, resources, and alias control. Classically, types in programming languages are used to check data formats, but we envisage much broader type-based verification, with specialised systems to analyse individual security properties. Indeed, Java 5 has annotations that support just such *pluggable* type systems [11].

Information flow. Work on information flow has focused on the definition of an accurate information flow type system for sequential Java bytecode and on its relation with information flow typing for Java source code, as well as on flexible analyses for concurrency.

Policies. Our work mainly focuses on termination insensitive policies which assume that the attacker can only draw observations on the input/output behavior of methods. Formally, the observational power of the attacker is captured by its security level (taken from a lattice $\mathbb{S}$ of security levels) and by *indistinguishability* relations $\sim$ on the semantic domains of the JVM memory, including the heap and the output value of methods (normal values or exceptional values).

Then, policies are expressed as a combination of global policies, that attach levels to fields, and local policies, that attach to methods identifiers signatures of the form $\boldsymbol{k_v} \xrightarrow{k_h} \boldsymbol{k_r}$, where $\boldsymbol{k_v}$ sets the security level of local variables, k_h is the heap effect of the method, and $\boldsymbol{k_r}$ is a record of security levels of the form $\{n : k_n, e_1 : k_{e_1}, \ldots e_n : k_{e_n}\}$, where k_n is the security level of the return value (normal termination) and each e_i is an exception class that might be propagated by the method, and k_{e_i} is its corresponding security level.

A method is safe w.r.t. a signature $\boldsymbol{k_v} \xrightarrow{k_h} \boldsymbol{k_r}$ if:

1. two terminating runs of the method with $\sim_{\boldsymbol{k_v}}$-equivalent inputs and equivalent heaps, yield $\sim_{\boldsymbol{k_r}}$-equivalent results and equivalent heaps;
2. the heap effect of the method is greater than k_h, i.e. the method does not perform field updates on fields whose security level is below k_h.

The definition of heap equivalence adopted in existing works on information flow for heap-based language, including [8], often assumes that pointers are opaque, i.e. the only observations that an attacker can make about a reference are those about the object to which it points. However, Hedin and Sands [29] have recently observed that the assumption is unvalidated by methods from the Java API, and exhibited a Jif program that does not use declassification but leaks information through invoking API methods. Their attack relies on the assumption that the function that allocates new objects on the heap is deterministic; however, this assumption is perfectly reasonable and satisfied by many implementations of the JVM. In addition to demonstrating the attack, Hedin and Sands show how a refined information flow type system can thwart such attacks for a language that allows to cast references as integers. Intuitively, their type system tracks the security level of references as well as the security levels of the fields of the object its points to.

Bytecode verification for secure information flow. We have defined a lightweight bytecode verifier that enforces non-interference of JVM applications, and proved formally its soundness against Bicolano [8]. The lightweight bytecode verifier performs a one-pass analysis of programs, and checks for every program point that the instruction verifies the constraints imposed by transition rules of the form

$$\frac{P[i] = ins \qquad constraints(ins, st, st', \Gamma)}{\Gamma, i \vdash st \rightarrow st'}$$

where i is an index consisting of a method body and a program point for this body, and the environment Γ contains policies, a table of security signatures for each method identifier, a security environment that maps program points to security levels, as well as information about the branching structure of programs, that is verified independently in a preliminary analysis. For increased precision, the preliminary analysis checks null pointers (to predict unthrowable null pointer exceptions), classes (to predict target of throw instructions), array accesses (to predict unthrowable out-of-bounds exceptions), and exceptions (to over-approximate the set of throwable exceptions for each method); the information is then used by a CDR checker that verifies control dependence regions (cdr), using the results of the PA analyser to minimise the size of regions.

Relation with information flow type system for Java. JFlow [34] is an information flow aware extension of Java that enforces statically flexible and expressive information policies by a constraint-based algorithm. Although the expressiveness of JFlow makes it difficult to characterize the security properties enforced by its type system, sound information flow type systems inspired from JFlow have been proposed for exception-free fragments of Java.

JFlow offers a practical tool for developing secure applications but does not address mobile code security as envisioned in MOBIUS since it applies to source code. In order to show that applications written in (a variant of) JFlow can be deployed in a mobile code architecture that delivers the promises of JFlow in terms of confidentiality, [7] proves that a standard (non-optimizing) Java compiler translates programs that are typable in a type system inspired from [5], but extended to exceptions, into programs that are typable in our system.

Concurrency. Extending the results of [8] to multi-threaded JVM programs is necessary in order to cover MIDP applications, but notoriously difficult to achieve. Motivated by the desire to provide flexible and practical enforcement mechanisms for concurrent languages, Russo and Sabelfeld [41] develop a sound information flow type system that enforces termination-insensitive non-interference in for a simple concurrent imperative language. The originality of their approach resides in the use of pseudo-commands to constrain the behavior of the scheduler so as to avoid internal timing leaks. One objective of the project is to extend their ideas to the setting of the JVM.

Declassification. Information flow type systems have not found substantial applications in practice, in particular because information flow policies based on non-interference are too rigid and do not authorize information release. In contrast, many applications often release deliberately some amount of sensitive information. Typical examples of deliberate information release include sending an encrypted message through an untrusted network, or allowing confidential information to be used in statistics over large databases. In a recent survey [42], A. Sabelfeld and D. Sands provide an overview of relaxed policies that allow for some amount of information release, and a classification along several dimensions, for example who releases the information, and what information is released. Type-based enforcement mechanisms for declassification are presented in [12].

Resource analysis. In §4.1 we identified requirements for MOBIUS resource security policies, as well as some notions of "resource" relevant to the MIDP application domain. Here we survey work within the project on analyses to support such policies, with particular focus on the possibility of formally verifying their correctness: essential if they are to be a basis for proof-carrying code.

Memory usage. The Java platform has a mandatory memory allocation model: a stack for local variables, and an object heap. In [9] we introduce a bytecode type system for this, where each program point has a type giving an upper limit on the number of heap objects it allocates. Correctness is proved via a translation into the MOBIUS logic, and every well-typed program is verifiable [21, Thm. 3.1.1]. Using the technique of *type-preserving compilation* we can lift this above the JVM: we match the translation from a high-level program F to bytecode $[\![F]\!]$ with a corresponding translation of types; and again for every well-typed program its bytecode compilation is verifiable in the MOBIUS logic [21, Thm. 3.1.3]. Even without the original high-level source program and its types, this low-level proof can certify the bytecode for PCC.

Work in the MRG project [4] demonstrated more sophisticated space inference for a functional language, using Hofmann-Jost typing [30] to give space bounds dependent on argument size, and with these types used to generate resource proofs in a precursor of the MOBIUS logic. We have now developed this further, into a space type system for object oriented programming based on amortised complexity analysis [31].

Billable events. Existing MIDP security policies demand that users individually authorise text messages as they are sent. This is clearly awkward, and the series of confirmation pop-up screens is a soft target for social engineering attacks. We propose a Java library of *resource managers* that add flexibility without compromising safety[21, §3.3]: instead of individual confirmation, a program requests authorisation in advance for a series of activities. Resource security may be assured either by runtime checks, or a type system for resource accounting, such that any well-typed program will only attempt to use resources for which it already has authorisation.

We have also used abstract interpretation to model such external resources [10]. From a program control-flow graph, we infer constraints in a lattice of permissions: whenever some resourceful action takes place, the program must have acquired at least the permissions required. Automated constraint solving can then determine whether this condition is satisfiable.

Execution time. Static analysis to count instructions executed can be verified in bytecode logic using *resource algebras* [3]. We have recently developed a static analysis framework [**?**] which provides a basis for performing cost analysis directly at the bytecode level. This allows obtaining cost relations in terms of the size of input arguments to methods. In addition, platform-dependent factors are a significant challenge to predicting real execution time across varied mobile platforms. We have shown how parameterised cost models, calibrated to an individual platform by running a test program, can predict execution times on different architectures [33]. In a PCC framework, client devices would map certified platform-independent cost metrics into platform-dependent estimates, based on fixed calibration benchmarks.

Alias control. Alias characterisations simplify reasoning about programs [26]: they enable modular verification, facilitate thread synchronisation, and allow programmers to exchange internal representations of data structures. Ownership types [18,17] and Universe types [35] are mechanisms for *characterising* aliasing in object oriented programming languages. They organise the heap into a hierarchical structure of nested non-overlapping *contexts* where every object is contained in one such context. Each context is characterised by an *object*, which is said to *own* all the objects contained directly in that context. Figure 3 illustrates the ownership structure of a linked list with iterator.

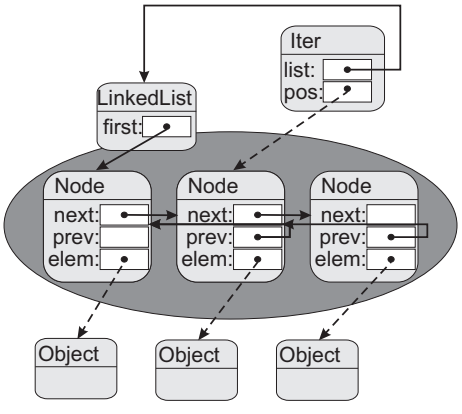

Fig. 3. Object structure of a linked list. The `LinkedList` object owns the nodes of the doubly-linked list. The iterator is in the same context as the list head. It has a `peer` reference to the list head and an `any` reference to the `Node` object at the iterator position.

In the Universe Type System [35,26], a context hierarchy is induced by extending types with Universe annotations, which range over `rep`, `peer`, and `any`. A field typed with a Universe modifier `rep` denotes that the object referenced by it must be within the context of the current object; a field typed with a Universe modifier `peer` denotes that the object referenced by it must be within the context that also contains the current object; a field typed with a Universe modifier `any` is agnostic about the context containing the object referenced by the field.

So far, we have concentrated on the following three areas:

- *Universe Java:* The formalisation and proof of soundness of a minimal object-oriented language with Universe Types.
- *Generic Universe Java:* The extension of Universe Java to Generic Java.
- *Concurrent Universe Java:* The use of Universe Types to administer race conditions and atomicity in a concurrent version of Universe Java.

UJ - Universe Java. As a basis for the other two work areas, we formalized Universe Java and proved the following key properties:

- *Type safety:* The Universe annotations `rep` and `peer` correctly indicate the owner of an object.
- *Encapsulation:* The fields of an object can only be modified through method calls made on the owner of that object (owner-as-modifier discipline).

GUJ - Generic Universe Java. We extended Universe Java to handle generics, which now form part of the official release of Java 1.5. In Generic Java, classes have parameters which can be bound by types: since in Universe Java, types are made up of a Universe modifier and a class, GUJ class parameters in generic class definitions are bound by Universe modifiers *and* classes. Generic Universe Java provide more static type safety then Universe Java by reducing the need for downcasts with runtime ownership checks. We proved that GUJ is type safe and enforces encapsulation.

UJ and Concurrency. The Universe ownership relation in UJ provides a natural way to characterise non-overlapping nested groups of objects in a heap. We therefore exploit this structure in a Java with multiple concurrent threads [25] to ensure atomcity and absence of data races.

6 Towards Certificate Generation and Certificate Checking

An important part of a PCC infrastructure is concerned with certificates. For the code producer one of the main tasks is to generate a certificate ensuring that his program meets the security policy of the client. In contrast, the code verifier/consumer needs to convince himself that the transmitted program respects his security policy.

In the scenario of Fig. 2 we assume that operators send compiled code, i.e. bytecode, to their customers, but this leaves the question of whether code producers will supply source code or bytecode to the operator. In MOBIUS, we concentrate on the latter, since this avoids the inclusion of the compiler in the trusted code base and does not require code producers to provide access to their source code.

6.1 Certificate Generation

The MOBIUS project focuses on two approaches for the generation of certificates, logic-based verification and type-based verification. By exploring both approaches, we hope to complement the rigorousness of our formalization by flexibility and automation.

The first technique (*logic-based verification*) is the concept of a proof transforming compiler [6], where properties can be specified and verified at the source code level and are then guaranteed to be preserved by the compilation, analogously to the way that *type-preserving compilation* guarantees the preservation of properties in the context of type systems. In addition to a program written in the source language, such a compiler expects a proof that the source program satisfies a (high-level) specification. Its output consist of the bytecode program and a proof (*certificate*) that this program satisfies the translation of the original specification into a formalism appropriate for bytecode. Logic-based verification is particularly suitable for functional correctness properties, but we have already shown in previous work how to generate JML annotations for a large

class of high-level security properties [39]. Interactive usage of the proof assistant, for example in order to discharge side conditions emitted by the VCgen, is also admissible. To be able to write such a proof transforming compiler for Java programs annotated with JML, we have developed a dedicated annotation language for Java bytecode: the Bytecode Modeling Language (BML) [13].

The second technique for the generation of specifications and certificates, *type-based verification*, rests on automated (and in general conservatively approximate) program analysis. Here, certificates are derived from typing derivations or fixed-point solutions of abstract interpretations, as outlined in the previous section and in the philosophy of lightweight bytecode verification.

6.2 Certificate Checking

For the code verifier/consumer, the goal is to check that the received program meets its specification (i.e. check the validity of the certificate) and to ensure that the specification is compliant with his security policies. Both parts should be fully automatic, and the machinery employed for this task is part of the trusting computing base (TCB).

The size of TCB is one of the main difficulties in a PCC architecture. Foundational PCC [2] minimizes the TCB by modeling the operational semantics of the bytecode in a proof assistant, and by proving properties of programs w.r.t. the operational semantics. Then deductive reasoning is used to encode program logic rules or typing rules. FPCC allows to remove the VCgen and type checkers for the application type systems from the TCB, but the deductive reasoning to encode proof rules or typing rules leads to bigger certificates than using a VCgen or a type checker.

One ambitious goal is to merge both approaches, and to get a small TCB and small certificates. Ultimately, a MOBIUS certificate is always a Coq proof of desired property phrased in terms of semantics. Apart from the proof assistant itself, Bicolano represents the trusting computing base of MOBIUS reasoning infrastructure. By representing formal systems in a proof assistant, we firstly increase the confidence in the validity of our checkers. Secondly, these representations allow us to exploit the infrastructure of the proof assistant when verifying concrete programs and their certificates.

Based on this, and complementing FPCC, the following two proof methodologies for type-based verification are considered within MOBIUS.

Derived Assertions. The Derived Assertions-Approach pioneered in MRG associates with each typing judgement an assertion in the program logic, the derived assertion. For each (schematic) typing rule one then proves a derived program logic proof rule operating on these derived assertions and possibly involving semantic, e.g. arithmetic, side conditions to be discharged by the proof assistant. Given a concrete typing derivation, a proof of the derived assertion corresponding to its conclusion can then be obtained by a simple tactic which invokes these derived rules mirroring the typing derivation. The typing derivation itself will typically be obtained using an automatic type inference which then need not be part of the TCB.

Reflection. Recent versions of Coq come with a powerful computational engine [27] derived from the OCAML compiler. This allows computationally intensive tasks to be

carried out within the proof assistant itself. A prominent example thereof is Gonthier-Werner's self-contained proof of the four-color theorem within Coq. This feature can be harnessed for our purposes in the following way using the reflection mechanism:

- we encode a type system T as a boolean-valued function $\mathsf{typable}_T$ on programs, and prove that the type system is sound in the sense that it enforces some expected semantic property interp_T. Formally, soundness is established by proving the lemma

$$\mathsf{TypeCorrect} : \forall P : \mathsf{prog}.\ \mathsf{typable}_T(P) = \mathsf{true} \implies \mathsf{interp}_T(P)$$

- to prove that $\mathsf{interp}_T(P_0)$ holds for a particular program P_0, we just have to apply the $\mathsf{TypeCorrect}$ lemma, and prove that $\mathsf{typable}_T(P_0) = \mathsf{true}$ holds.
- if your checker allows you to reason by computation (i.e. two propositions are equal if they are computationally equal) and if the program P_0 is typable, the proposition

$$\mathsf{typable}_T(P_0) = \mathsf{true}$$

  is equal (i.e. reduces) to $\mathsf{true} = \mathsf{true}$ which is trivial to prove.

The Coq proof assistant allows such a reasoning mechanism. In Coq, the representation of such a proof is $\mathsf{TypeCorrect}\ P\ (\mathsf{refl\_equal}\ \mathsf{true})$, where $(\mathsf{refl\_equal}\ \mathsf{true})$ is a proof of $\mathsf{true} = \mathsf{true}$.

Similar to this reflectional approach to PCC is the technique we presented in [14], where lattice abstract interpretation is used to verify bounded memory use. Significantly, here both the algorithm and its correctness proof are expressed within the Coq proof assistant, such that we may extract a *certified checker* from the proof itself. This allows a novel realisation of proof-carrying code, where a fast program verifier is trusted because it is obtained from its own proof of correctness.

7 Next Steps

After a year activity, the MOBIUS project is well on tracks. Scientific progress is proceeding as expected: security requirements and the PCC scenarios for global computing have been defined, and significant advances in enabling technologies have been reported in deliverables and scientific publications. For further information, please consult `http://mobius.inria.fr`.

References

1. Albert, E., Puebla, G., Hermenegildo, M.V.: Abstraction-carrying code. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS, vol. 3452, pp. 380–397. Springer, Heidelberg (2004)
2. Appel, A.W.: Foundational proof-carrying code. In: Halpern, J. (ed.) Proceedings of the Sixteenth Annual IEEE Symp. on Logic in Computer Science, LICS 2001 (Invited Talk), p. 247. IEEE Computer Society Press, Los Alamitos (2001)

3. Aspinall, D., Beringer, L., Hofmann, M., Loidl, H.-W., Momigliano, A.: A program logic for resource verification. In: TPHOLs 2004. LNCS, Springer, Heidelberg (2004)
4. Aspinall, D., Gilmore, S., Hofmann, M., Sannella, D., Stark, I.: Mobile Resource Guarantees for Smart Devices. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 1–27. Springer, Heidelberg (2005)
5. Banerjee, A., Naumann, D.: Stack-based access control for secure information flow. Journal of Functional Programming (Special Issue on Language-Based Security) 15, 131–177 (2005)
6. Barthe, G., Grégoire, B., Kunz, C., Rezk, T.: Certificate translation for optimizing compilers. In: SAS'06: Proceedings of Static Analysis Symposium. LNCS, Springer, Heidelberg (2006)
7. Barthe, G., Naumann, D., Rezk, T.: Deriving an information flow checker and certifying compiler for java. In: Symposium on Security and Privacy, 2006, IEEE Press, Orlando (2006)
8. Barthe, G., Pichardie, D., Rezk, T.: A certified lightweight non-interference java bytecode verifier. In: Niccola, R.D. (ed.) Proceedings of ESOP'07. LNCS, vol. 4xxx, Springer, Heidelberg (2007)
9. Beringer, L., Hofmann, M.: A bytecode logic for JML and types. In: Kobayashi, N. (ed.) APLAS 2006. LNCS, vol. 4279, pp. 389–405. Springer, Heidelberg (2006)
10. Besson, F., Dufay, G., Jensen, T.P.: A formal model of access control for mobile interactive devices. In: ESORICS 2006. LNCS, Springer, Heidelberg (2006)
11. Bracha, G.: Pluggable type systems. Presented at the OOPSLA 2004 Workshop on Revival of Dynamic Languages (October 2004)
12. Broberg, N., Sands, D.: Flow locks: Towards a core calculus for dynamic flow policies. In: Sestoft, P. (ed.) ESOP 2006 and ETAPS 2006. LNCS, vol. 3924, pp. 180–196. Springer, Heidelberg (2006)
13. Burdy, L., Huisman, M., Pavlova, M.: Preliminary design of BML: A behavioral interface specification language for Java bytecode. In: TSDM 2000. LNCS, Springer, Heidelberg (to appear)
14. Cachera, D., Jensen, D.P.T., Schneider, G.: Certified memory usage analysis. In: Fitzgerald, J.A., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 91–106. Springer, Heidelberg (2005)
15. Charles, J.: Adding native specifications to JML. In: ECOOP workshop on Formal Techniques for Java-like Programs (FTfJP'2006) (2006)
16. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 1–18. Springer, Heidelberg (2003), Available from `http://www.brics.dk/JSA/`
17. Clarke, D.G., Drossopoulou, S.: Ownership, Encapsulation and the Disjointness of Type and Effect. In: OOPSLA, pp. 292–310 (2002)
18. Clarke, D.G., Potter, J.M., Noble, J.: Ownership Types for Flexible Alias Protection. In: Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98), ACM SIGPLAN Notices, vol. 33(10), pp. 48–64. ACM Press, New York (1998)
19. Consortium, M.: Deliverable 1.1: Resource and information flow security requirements (2006), Available online from `http://mobius.inria.fr`
20. Consortium, M.: Deliverable 1.2: Framework-specific and application-specific security requirements (2006), Available online from `http://mobius.inria.fr`
21. Consortium, M.: Deliverable 2.1: Intermediate report on type systems (2006), Available online from `http://mobius.inria.fr`
22. Consortium, M.: Deliverable 3.1: Bytecode specification language and program logic (2006), Available online from `http://mobius.inria.fr`

23. Consortium, M.: Deliverable 4.1: Scenarios for proof-carrying code (2006), Available online from http://mobius.inria.fr
24. Crégut, P., Alvarado, C.: Improving the security of downloadable Java applications with static analysis. In: Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode 2005). Electronic Notes in Theoretical Computer Science, vol. 141, Elsevier Science, Inc, North-Holland (2005)
25. Cunningham, D., Drossopoulou, S., Eisenbach, S., Dietl, W., Müller, P.: CUJ: Universe Types for Race Safety. Preliminary version at http://slurp.doc.ic.ac.uk/pubs.html#cuj06
26. Dietl, W., Müller, P.: Universes: Lightweight ownership for JML. Journal of Object Technology (JOT) 4(8), 5–32 (2005)
27. Grégoire, B., Leroy, X.: A compiled implementation of strong reduction. In: ICFP'02: Proceedings of the International Conference on Functional Programming, pp. 235–246. ACM Press, New York (2002)
28. Hähnle, R., Mostowski, W.: Verification of safety properties in the presence of transactions. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 151–171. Springer, Heidelberg (2005)
29. Hedin, D., Sands, D.: Noninterference in the presence of non-opaque pointers. In: Proceedings of the 19th IEEE Computer Security Foundations Workshop, IEEE Computer Society Press, Los Alamitos (2006)
30. Hofmann, M., Jost, S.: Static prediction of heap space usage for first-order functional programs. In: POPL'03, Proceedings of the 30rd Annual. ACM SIGPLAN - SIGACT. Symposium. on Principles of Programming Languages, pp. 185–197. ACM Press, New York (2003)
31. Hofmann, M., Jost, S.: Type-based amortised heap-space analysis. In: Proceedings of ESOP2006, pp. 22–37 (2006)
32. U.T. Initiative Unified testing criteria for Java technology-based applications for mobile devices. Technical report, Sun Microsystems, Motorola, Nokia, Siemens, Sony Ericsson, Version 2.1 (May 2006)
33. Mera, E., López-García, P., Puebla, G., Carro, M., Hermenegildo, M.: Combining Static Analysis and Profiling for Estimating Execution Times. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, Springer, Heidelberg (2006)
34. Myers, A.: JFlow: Practical mostly-static information flow control. In: Myers, A. (ed.) POPL'99, Proceedings of the 26rd Annual. ACM SIGPLAN - SIGACT. Symposium. on Principles of Programming Languages, pp. 228–241. ACM Press, New York (1999)
35. Müller, P.: Modular Specification and Verification of Object-Oriented Programs. PhD thesis, FernUniversität Hagen (2001)
36. Necula, G.C.: Proof-carrying code. In: POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages, pp. 106–119. ACM Press, New York (1997)
37. Necula, G.C., Lee, P.: Safe kernel extensions without run-time checking. In: Proceedings of Operating Systems Design and Implementation (OSDI), Seattle, WA, USENIX Assoc, pp. 229–243 (October 1996)
38. Parliement, E., Council, E.: Directive 95/46/ec of the european parliament and of the council of 24 october 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data. Official Journal of the European Communities, number L 281, 31–50 (october 1995)
39. Pavlova, M., Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L.: Enforcing high-level security properties for applets. In: Paradinas, P., Quisquater, J.-J. (eds.) Proceedings of CARDIS'04, Toulouse, France, August 2004, Kluwer Academic Publishers, Boston (2004)

40. Rose, E.: Lightweight bytecode verification. Journal of Automated Reasoning 31(3-4), 303–334 (2003)
41. Russo, A., Sabelfeld, A.: Securing interaction between threads and the scheduler. In: Proceedings of CSFW'06 (2006)
42. Sabelfeld, A., Sands, D.: Dimensions and principles of declassification. In: Proceedings of CSFW'05, IEEE Press, Orlando (2005)
43. The Coq development team. The coq proof assistant reference manual v8.0. Technical Report 255, INRIA, France, mars (2004), `http://coq.inria.fr/doc/main.html`