INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Project-Team EVEREST

Vérification et sécurité du logiciel

Sophia Antipolis

THEME SYM

Activity Report

2006

Table of contents

1. Team

Team Leader
Gilles Barthe [Research Director INRIA, HdR]

Team Vice-Leader
Marieke Huisman [Research scientist INRIA]

Research scientist
Benjamin Grégoire [Research scientist INRIA]

Administrative assistant
Nathalie Bellesso

Ph.D. student
Julien Charles [MESR grant, Teaching Assistant UNSA]
Clément Hurlin [since September 2006]
Allard Kakebeen [until February 2006]
César Kunz
Mariela Pavlova [until October 2006]
Tamara Rezk [until September 2006]
Sabrina Tarento [until September 2006, MESR grant, Teaching Assistant UNSA]
Santiago Zanella [since June 2006]

Post-doctoral Fellow
Julien Forest [until September 2006]
Romain Janvier [since September 2006]
Yu Zhang [since November 2006]

Technical Staff
Sophie Hadjadj
Anne Pacalet [since January 2006]
David Pichardie [until August 2006]

Visiting Ph.D. student
Salvador Cavadini [visiting Ph.D. student from University of San Luis, Argentina, since February 2006, 18 months]

Student internship
Fernando Pastawski [Cordoba University, Argentina, 6 months]
Gustavo Petri [Cordoba University, Argentina, 6 months]
Alejandro Tamalet [Rosario University, Argentina, 5 months]

2. Overall Objectives

2.1. Overall Objectives

The Everest project concentrates on increasing reliability and security of mobile and embedded software. This is achieved by developing and applying formal methods and language-based techniques, covering both platform and application level. The project's privileged application domain ranges from trusted personal devices, such as mobile phones and smart cards, to ubiquitous computing.

The project focuses on the following research areas:

- Program verification and Proof Carrying Code;
- Machine-checked semantics and algorithms;
- Foundations of proof assistants.

3. Scientific Foundations

3.1. Type systems

Types are often considered as one of the great successes of programming language theory, and their use permeates modern programming languages. The widespread use of types is a consequence of three crucial factors:

- *Types are intuitive*. Types are a particularly simple form of assertion. 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. For example, Fortran and C use type systems to generate memory layout directives at compile time; yet the users of C can write type-correct programs and understand typing errors without knowing exactly how the type concept is related to memory layout.
- *Types are automatic*. In many cases an untyped program can be enhanced with types automatically by type inference. Of course, 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 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 and simplicity 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.

Type systems have long been used in programming languages to enforce basic safety properties of programs. For example, the type system of Java is designed to statically detect certain runtime errors such as the application of a string function to a floating point number, or a call to a method that does not belong to a name space of a given class. In addition, the research community has developed numerous type systems that enforce more advanced safety properties dealing with modularity, concurrency, and aliasing in Java programs.

Type systems are also increasingly being studied as a means to enforce security; in particular, the research community has developed type systems that guarantee secure information flow and resource control policies.

3.2. Program verification

Research on program logics has a long history, dating back to the seminal work on Floyd-Hoare logics and weakest precondition calculi in the late 1960s and early 1970s. Although this line of research has not yet lead to a breakthrough in the application of program verification, there has been steady progress, resulting in tool-supported program logics for realistic programming languages.

There are a number of reasons to adopt program verification techniques based on logic to guarantee the correctness of programs.

- *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 exists for describing computational systems.
- *Logic is precise*. While types generally over-approximate program behaviour, logic can be used to provide precise statements about program behaviour. Special conditions can be assumed and exceptional behaviours 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 analyses to be combined*. Logic provides a common setting into which the declarative content of a typing judgement 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.

Recently, there has been a lot of research into logic-based program verification of Java, which has culminated in the realisation of program verification environments for single-threaded Java.

3.3. Machine-checked semantics and algorithms

We are interested in developing formal, machine-checked semantics of programming languages and of their execution platforms such as virtual machines, run-time environments, Application Programming Interfaces, and of the tools that are used for compiling, verifying, validating programs. In particular, we have strong experience in modelling and verifying execution platforms for smart cards, as well as their main security functions, such as bytecode verifiers and access control mechanisms, and the standard deployment architectures for multi-application smart cards, such as Global Platform.

We are also interested in developing formal, machine-checked security proofs for cryptographic algorithms, using tools from provable cryptography. In particular, we have formalised the Generic Model and Random Oracle Model, and given formally verified security bounds for the probability of an attacker breaking the discrete logarithm and related encryption and signing schemes.

3.4. Software security

Security is not a technology, but a property of a system. For this reason, there are new security requirements associated with each new technology or architecture. While these security requirements are often expressed from the perspective of the users, they must also be translated to concrete objectives that can be enforced by security mechanisms.

For instance, the Java security model assumes that end users trust its runtime environment but not the downloaded code. The concrete objectives include adherence to the typing and policy of the JVM and compliance with the stack inspection mechanism, which are enforced by the Java byte code verifier and the runtime environment.

The ability to derive concrete verification objectives from carefully gathered security requirements is an important step for guaranteeing that a component is secure with respect to a given security policy. Typical requirements include information flow security policies; resource control policies; framework-specific security; and application-specific security.

We give precise mathematical definitions of these requirements, using programming language semantics, and provide means to enforce these requirements using type systems or logic.

3.5. Proof Carrying Code

Proof Carrying Code (PCC) is an innovative security framework in which components come equipped with a certificate which can be used by devices (code consumers in PCC terminology) to verify locally and statically that downloaded components issued by an untrusted third party (code producers in PCC terminology) are correct. In order to realise this view, standard PCC infrastructures build upon several elements: a logic, a verification condition generator, a formal representation of proofs, and a proof checker.

- *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. Standard PCC 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, properties to be satisfied by the state before and after a given procedure or function is invoked.
- *A verification condition generator (VCGen).* The VCGen 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. Standard PCC adopts a VCGen 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 Standard PCC, certificates are terms of the lambda calculus, as suggested by the Curry-Howard isomorphism, and routinely used in modern proof assistants such as Coq.
- *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 VCGen. In Standard PCC, proof checking is reduced to type checking by virtue of the Curry-Howard isomorphism. One very attractive aspect of this approach is that the proof checker, which forms part of the Trusted Computing Base is particularly simple.

We study other variants of Proof Carrying Code that cover a wide range of security properties that escape the scope of certifying compilers, and that need to be established interactively on source code programs.

4. Application Domains

4.1. Smart devices

Smart devices, including new generation smart cards and trusted personal devices typically contain a microprocessor and a memory chip (but with limited computing and storage capabilities). They are often used by commercial and governmental organisations and are expected to play a key role in enforcing trust and confidence in e-payment and e-government. Current applications include bankcards, e-purses, SIM cards in mobile phones, e-IDs, *etc*,

These devices provide solutions for application developers by enabling them to program in high-level languages (several dialects of Java exist for this purpose: Java Card, MIDP, *etc.*), on a common software base (a virtual machine and Application Programming Interfaces, APIs), which isolates their code from specific hardware and operating system libraries. These devices support both the flexibility and the evolution of applications by enabling downloading of executable content onto already deployed devices (so-called post issuance), and by allowing several commercially independent applications to run on a single device. This open character forms their commercial strength, but also creates a technical challenge: reliability, correctness and security become crucial issues, since malicious applications might potentially exploit bugs in the smart device platform, with detrimental effects on security and/or privacy.

4.2. Global computing

A global computer is a *distributed* computational infrastructure that aims at providing a global and uniform access to services. However, global computers consist of large networks of *heterogeneous* devices that differ greatly in their computational infrastructure and in the resources they offer to services. In order to deliver services globally and uniformly, each device in a global computer must therefore be *extensible* with the computational infrastructure, platform or libraries, needed to execute the required services. In that respect, global computers transcend the scope of established computational models such as mobile code, the Grid, or agents, which impose a clear separation between mobile applications and the fixed computational infrastructure upon which they execute.

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 or e-health, involve sensitive data that must be protected from unauthorised parties. 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 of the operator. Other models, building upon 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 do not embrace the complexity of global computers.

5. Software

5.1. Jack: a tool for applet validation

Participants: Gilles Barthe, Julien Charles, Benjamin Grégoire, Marieke Huisman, Mariela Pavlova.

In 2003, the team took over the development of Jack (Java Applet Correctness Kit, a tool for applet validation), initiated within the research laboratory of the smart card manufacturer Gemplus (now Gemalto). The work was originally carried out in the context of an ODL, with Lilian Burdy, the chief architect of Jack, working as a research engineer within the ODL.

The original motivation for Jack was to provide a developer-oriented environment to validate Java smart card applications annotated with JML annotations; the main emphasis of the original release was in hiding the complexity of formal verification using automatic provers, and in enabling developers to use formal verification technology within a standard IDE, namely Eclipse. Since 2004, we have added many features to Jack:

- A plug-in to prove proof obligations interactively using the Coq proof assistant, as well as an interface to use Coq within Eclipse. Together with the Coq plugin, we also developed several Coq tactics to solve or simplify some proof obligations automatically (and therewith increase their readability) [24].

  A similar Coq plug-in has been developed for ESC/Java.
- A plug-in to generate annotations from high-level security properties [27], and another plugin to generate annotations to prevent nullpointer and array-index-out-of-bound exceptions, and to generate modifies clauses.
- A plug-in (JML2BML) to translate JML specifications into BML specifications (BML, the Bytecode Modeling Language, is the bytecode cousin of JML, developed by the team), and another plug-in (BML-VcGen) to generate verification conditions from Java bytecode and its BML specification [9].
- Support for a `native` keyword in JML, to enable connecting a JML specification with the logic of the underlying prover. This makes it easier to reason about complicated data structures [10].

The Jack tool has been used in small to medium-scale experiments. The most conclusive application, carried out in collaboration with researchers from the POPS team at INRIA Futurs and researchers from Gemalto, used the verification tool at bytecode level to reduce the footprint of Java-to-native compilation schemes [11].

Most of the work on Jack has been conducted within the European project Inspired [20]. A survey article [21] is in preparation for the proceedings of FMCO'06.

5.2. Jakarta

Participants: Gilles Barthe, Julien Forest.

We develop a tool for formally specifying and verifying execution platforms, and more particularly bytecode verifiers, as they exist *e.g.* in the Java Platform.

The tool, which instruments a two-phase methodology that is common to many existing works, intends to provide a very high-level of automation for the mundane tasks inherent to the methodology (namely deriving a virtual machine from another by abstraction techniques, and proving the correctness of the abstraction), and has been used successfully for certifying the correctness of the Java Card bytecode verifier.

The development of the tool was initiated in the context of the European project Verificard and is continued partly in the context of the RNTL project CASTLES which focuses on the certification of execution platforms for smart cards.

6. New Results

6.1. Program verification and Proof Carrying Code

Participants: Gilles Barthe, Salvador Cavadini, Julien Charles, Benjamin Grégoire, Marieke Huisman, Clément Hurlin, Romain Janvier, César Kunz, Anne Pacalet, Mariela Pavlova, Gustavo Petri, David Pichardie, Tamara Rezk, Alejandro Tamalet, Santiago Zanella, Yu Zhang.

1. We studied certificate translators for optimising compilers, and in particular the means to transform certificates of source code into certificates of executable code in the context of an optimising compiler [6]. Currently we are working on an implementation of a certificate translator in OCAML. In addition, we have begun studying certificate translation for aspect-oriented languages.
2. We have defined an information flow type system for a sequential JVM-like language that includes classes, objects, arrays, exceptions and method calls, and we have proven that it guarantees non-interference [18], [2]. For increased confidence, we have formalised the proof in the proof assistant Coq, showing soundness of the type system *w.r.t.* the Bicolano semantics (see below); an additional benefit of the formalisation is that from our proof we have extracted a certified lightweight bytecode verifier for information flow. Our work provides, to the best of our knowledge, the first sound and implemented information flow type system for such an expressive fragment of the JVM.

   In addition, we have developed a systematic technique to connect source code and bytecode security type systems [7], [2], and we have applied this technique to the information flow type system with exceptions, mentioned above. This is a key step towards the deployment of language-based security in practical applications.
3. Development and maintenance of Jack has continued this year. The bytecode verification framework was used in a case study, carried out in collaboration with researchers from the POPS team at INRIA Futurs and researchers from Gemalto, where proven absence of possible exceptions is used to reduce the footprint of Java-to-native compilation schemes [11].

   To make interactive verification simpler, an editor for Coq has been integrated into Eclipse. This editor provides basically the same functionalities as CoqIde, but allows the developer to stay in the same environment both for development and verification of the application.
4. To improve Jack's support for interactive verification, we developed two example applications: an implementation of quicksort, as well as a Java bytecode verifier.

   The quicksort algorithm is implemented as a single method. It has been fully annotated and proved interactively. This proof could not have been done automatically, because of the extensive use of loop invariants and ghost variables.

   The bytecode verifier is a fixpoint algorithm: each instruction is associated with an abstract memory state, and the main loop iterates on the instruction graph to refine the information we have on each state. We have proved that for each iteration of the main loop the information on the states is refined, and we also proved several properties concerning the exception-freeness of the different functions.

   However, so far we could not prove termination of the algorithm, because we needed a transformation function from states to integers (in JML, the sole available termination argument possible is an integer). Since this transformation function is (too) difficult to define using only Java and JML annotations, we introduced the means to bind prover-native implementations to JML specifications of methods and types. For this, we introduced a new keyword in JML, the native keyword [10]. The main target of the native keyword is static verification, especially in order to use prover-specific features, but it can also be used for run-time verification. If a whole theory is defined within a specific prover, it can be bound easily to JML annotations using the keyword native.
5. One of the results of our work on the verification of bytecode is the definition of the Bytecode Modeling Language (BML), the bytecode cousin of JML [9], [8]. Within the Mobius project, a

specific task force has been identified to write a reference manual for BML. This reference manual will specify the syntax and semantics of the language, the class file format that will be used to store the annotations, and the compiler from JML to BML. This will serve as a basis for the tool development of the bytecode subcomponent of the Mobius tool set. The writing of the reference manual is currently starting; it will be available via http://www-sop.inria.fr/everest/BML. The team is one of the active partners in the specific task force.

6. An algorithm to generate annotations for high-level security properties has been implemented in Jack [27]. We are currently formalising this algorithm, in order to prove formal correctness of the approach. Given a property described by a finite state machine (FSM) and a Java application, we have defined how the FSM can be captured by appropriate JML set annotations. We have defined a "monitoring semantics", describing how the transitions in the FSM are triggered by method calls in the Java application. The FSM is extended in such a way that if it cannot make a transition, it will end up in a sink-state. For the Java application, we add an invariant that this sink-state should never be reached, and we prove that if the monitoring semantics does not reach the sink-state, this invariant will not be violated. Currently, we are proving that if the monitoring with the FSM will not find any errors, then runtime assertion checking will not find any assertion violations. A next step is to propagate the annotations, and to show that eventually correctness of the annotations can be proven statically.

7. We have been investigating ways to extend specification languages to cope with multi-threading. Initial work [28] identified a promising approach by identifying notions as atomicity, independence and immutability, that allow us to use existing methods and tools for verification of single-threaded code, by separating the multi-threaded aspects from the functional aspects of the specification. We have started working on formal definitions of the concepts involved. This showed us that the notions as proposed by Rodriguez *et al.* are too restrictive, as they do not take the actual method specification into account. We are currently working on overcoming these limitations, defining a precise semantics, and then appropriate proof rules for verifying them.

8. We are studying ways of combining static program analysis techniques to simplify formal program verification. We defined a *domain-based program decomposition* and proved that if a postcondition holds for each element in the decomposition, then the postcondition holds for the complete program. We think that domain-based decomposition is a powerful tool to simplify the verification of programs, because it divides the program into less complex subprograms. We use program slicing techniques to find the decomposition, *i.e.* to get the subprograms.

   In addition, we defined a new slicing technique, called *reachability slicing*. This new technique gives an answer to the following question: *which sentences are (potentially) executed when sentences* $s_1, s_2, ...,$ *and* s_n *are executed?* We defined how reachability slices can be computed with the help of other slicing techniques and how they then can be combined to obtain better, *i.e.* smaller, slices.

   These two results, reachability slicing and domain-based decomposition, are related. Reachability slicing allows us to use program sentences as decomposition criteria; which makes it easy to formulate and perform the decomposition.

9. Within the context of the Mobius project, we have contributed to two deliverables that define the security requirements for the Mobius security architecture. We contributed to the identification of three kinds of security requirements: information flow policies, framework-specific policies and application-specific policies. We addressed in particular the following topics:

   1. how existing information flow policies can be refined for bytecode;
   2. how framework-specific security requirements can by expressed using existing specification languages, in particular JML;
   3. what are typical application-specific security requirements for operating system components, and in particular for a bytecode verifier.

10. In collaboration with D. Gurov and I. Aktug from KTH, Sweden, we have continued our work on adapting our compositional verification techniques [19] to more elaborate program models. The basic program model that we used to develop our compositional verification techniques is a control flow graph of the application, where nodes are control points, possibly labelled as a return point, and edges are labelled as either internal actions or method calls. To include exceptions in the program model, we added explicit throwing and catching of exceptions. Each control point can be labelled with an exception, denoting that the program is an exceptional state. The behaviour of method calls is changed: the callstack contains a set of possible return points, and depending on whether the return state is normal or exceptional, an appropriate return point is selected. We have shown that with this modification our compositional verification techniques still apply. We combined this with the extension for multi-threading that we have developed earlier.

    In addition, we also looked at how we can characterise behavioural properties by sets of structural properties. Our compositional verification techniques only apply when the local assumptions are structural. When a behavioural property can be characterised by a set of structural properties, this means that we can also have behavioural properties as local assumptions. We have shown how modal logic behavioural formulae can be characterised by structural formulae, and we are currently working on extending this with greatest fixpoints. This requires a tableau construction and the use of a global discharge rule.

11. No matter how carefully crafted cryptographic primitives and protocols are, experience has shown that effective attacks can remain hidden for years and have disastrous consequences when made public. It is agreed that the point has been reached where it is no longer viable to construct cryptographic proofs by hand, nor even verify them. Shoup [29], Bellare and Rogaway [23], and Halevi [25] advocate the construction of cryptographic proofs as sequences of probabilistic games as a natural solution for taming the complexity of the task. They also recognise that a fully-specified programming language is required to code those games and that language-based techniques are needed for manipulating them. To answer these concerns, we have designed a probabilistic imperative programming language and formalised its semantics in the logic of the Coq theorem prover. This permits us to fully specify cryptographic proofs and gives a rudimentary way to verify them by reasoning directly about their semantics. Next, we plan to devise a framework to ease the construction of the proofs. By providing and building upon a Hoare-like logic and a weakest precondition calculus we expect to automate most mundane aspects of the construction, leaving the creative part of the task to be carried out interactively by the user.

6.2. Machine-checked semantics and algorithms

Participants: Gilles Barthe, Julien Forest, Benjamin Grégoire, Marieke Huisman, Allard Kakebeen, Mariela Pavlova, David Pichardie, Gustavo Petri, Sabrina Tarento.

1. We have completed our models of the bytecode verifier. The syntax of the Jakarta Specification Language (JSL) has been extended with both record types and polymorphic higher order types. These extensions required the development of a completely new typing algorithm for JSL which has been implemented in Jakarta. A basic system module has been developed and implemented. This leads to a considerable simplification of formal proofs and code (due in particular to the development of a standard library for basic datatypes).

2. We are in charge of the maintenance and development of Bicolano, which is a formal semantics in Coq of a representative fragment of the Java bytecode language. In the Mobius project, this semantics is used as a reference for all the formal developments around Java bytecode. This year, we have especially worked on the efficient implementation of several of the interfaces of the Bicolano development (in particular concerning program syntax and semantic domains).

3. We have pursued our work on the formalisation of the generic model (GM) and the random oracle model (ROM) in COQ. The aim of this work is to verify the correctness of cryptographic algorithms,

without making the perfect cryptography assumption. In [30], we considered the case of parallel attacks, for which we have shown an upper bound to the probability of an interactive adversary to make a one-more signature forgery (assuming that the adversary adheres to the GM and to the ROM *i.e.*, he can make interactions with a hash oracle and a signature oracle).

4. We have participated in the development of the so-called Mobius base logic, a program logic for Java bytecode, and to the development of its correctness proof. In Coq, this logic is proved sound *w.r.t.* the Bicolano semantics. We have also developed a verification condition generator (VCgen) for this logic and proved its correctness (also in Coq). We are currently working on several optimisations for this VCgen; we look in particular at different ways to reduce the control flow graph of the program to limit the number of proof obligations generated.
5. Currently there are two kinds of verification condition generators. The first approach propagates all information it has (derived from the annotations) to the entry point of the program. To implement such a VCgen, extra information is required (given by for example the *loop modifies* clause, that specifies which variables may be modified by a loop). In the second approach, for each annotation a verification condition is generated (specifying that the annotation should imply the weakest precondition of its successor). The advantage of the first approach is that the required annotations are smaller (so the task of the programmer is simpler), where the advantage of the second approach is it simplicity. In particular, to implement and prove the correctness of a VCgen for bytecode using the first approach is an order of magnitude more complex than using the second approach.

   We have implemented and proved correct an algorithm that automatically transforms annotations (including loop modifies clauses) for a VCgen using the first approach into annotations for a VCgen using the second approach. The correctness proof is an algorithm to automatically transform proofs of the verification conditions generated by the first VCgen into proofs of the verification conditions generated by the second one.
6. We are currently formalising the new Java Memory Model (introduced in Java 1.5) [26]. This memory model corrects certain security leaks that were possible previously, and in addition allows many common compiler optimisations. The goal of our formalisation is to prove that in case the program is correctly synchronised (meaning that it does not contain race conditions), the set of legal program behaviours coincides with the behaviours described by an interleaving semantics (this property is claimed by the developers of the Java Memory Model—we will verify it formally). As we are only interested in verifying correctly synchronised programs, this allows us to assume an interleaving semantics to prove the correctness of an application.

   We are also planning to study whether we can formally prove that the new Java Memory Model does not contain security leaks, in particular that it cannot be used to forge variables *out of thin air*. Finally, we plan to study whether we can find alternative and more intuitive, but equivalent, formulations of the Java Memory Model.

6.3. Type theory and proof assistants

Participants: Gilles Barthe, Julien Forest, Benjamin Grégoire, Fernando Pastawski, David Pichardie.

1. We have developed a practical method to define and reason about general recursive functions in Coq [4]. This method allows the user to easily define and reason about not-structurally recursive functions in Coq. The method has been merged and extended with the method due to A. Balaa and Y. Bertot [22] and integrated into the Coq proof assistant. Currently, the method allows the user to define a large class of well-founded functions directly (using either *well-founded induction* or a *well-founded measure*), while structural and non-recursive functions can be defined via the *Function* keyword. In the case of well-founded or measured functions, the user has to prove some proof obligations to ensure that the function is terminating. In all cases, the system automatically derives an *induction principle*, a *fixpoint equation*, a *graph*, and an equivalence proof between graph and function. In the case of structural functions, one can also define mutually recursive functions and automatically derive mutual inductive principles.

2. We have extended our type-based termination system [5] to the Calculus of Inductive Constructions. The system (CICˆ) is more powerful than the actual guard condition of Coq and easier to understand.
3. We participate to the development of the Coq proof assistant. In particular, we maintain and extend the conversion algorithm based on a compiler and a virtual machine (which has been recently added to Coq). This year, a new compilation scheme, based on lazy evaluation, has been implemented for co-fixpoints. We also participate to the integration of the new *ring* and *field* tactics.
4. In collaboration with L. Théry (Marelle) and B. Werner (LogiCal) we have developed a reflexive procedure to prove the primality of large prime numbers [13] in Coq. This technique is based on Pocklington's theorem. We have also developed a special procedure for Mersenne numbers. To speed up the verification of such generated proofs we have (with L. Théry) implemented an efficient library for arbitrary precision arithmetic in Coq and proved its correctness [12].

7. Contracts and Grants with Industry

7.1. Contracts

RNTL Castles (Development of Static Analyses and Test for Secure Embedded Software, accepted in 2003, started January 2004). Other participants are Lande (Rennes), AQL and Oberthur. More information is available via http://www-sop.inria.fr/everest/projects/castles/. The goal of the project is to define an environment to support the formal specification and verification of low-level execution platforms.

CEA The CEA (Commissariat à l'Energie Atomique) develops static analysis tools for C programs based on techniques such as precondition computation using Hoare logic and abstract interpretation. The collaboration (2006-2009) aims to enhance these tools by adding slicing capabilities in order to help managing bigger applications. Building smaller models is especially useful to study applications that can not be fully handled by other analyses. Anne Pacalet and Salvador Cavadini are involved in this collaboration.

INRIA-Microsoft Research Joint Laboratory (Secure Distributed Computations and their Proofs, 2006-2009). Other participants are the Programming Principles and Tools group at Microsoft Research Cambridge and the Moscova team (Rocquencourt). This project intends to design formal tools for programming distributed computations with effective security guarantees. Gilles Barthe, Benjamin Grégoire, and Santiago Zanella are involved in the project, see http://joint-lab.futurs.inria.fr/.

7.2. National projects

ACI Security GECCOO (Generation of Certified Code for Object-Oriented Applications - Specifications, refinement, proof and error detection, 2003-2006). Other participants are TFC (Besançon), Cassis (Nancy), ProVal (LRI/Futurs) and Vasco (IMAG, Grenoble), see http://geccoo.lri.fr/.

ACI Security SPOPS (Secure Operating Systems for Trusted Personal Devices, 2003-2006). Gilles Barthe was the coordinator of this project that involved the POPS team (INRIA Lille) and SSIR (Supélec, Rennes), see http://www-sop.inria.fr/everest/projects/spops/.

7.3. European projects

Mobius Gilles Barthe is the scientific coordinator of the European Integrated Project Mobius (Mobility, Ubiquity and Security), launched under the FET Global Computing Proactive Initiative in the 6th Framework program (2005-2009). The project gathers 12 academic and 4 industrial partners. The goal of this project is to develop the technology for establishing trust and security for the next generation of global computers, using the Proof Carrying Code paradigm. The essential features of the Mobius security architecture will be innovative trust management, static enforcement mechanisms and support for system component downloading, see http://mobius.inria.fr.

Inspired (2003-2006): The partners in the project are all major industrial actors in the domain of smart cards, INRIA (also the projects Metiss and POPS) and the Universities of Louvain (Belgium) and Twente (Netherlands). The goal of the project is to define the new generation of Trusted Personal Devices. The role of INRIA is to develop appropriate formal methods for those. Gilles Barthe is Work-Package leader for the Development Tools and Methodologies, see http://www.inspiredproject.com/.

Thematic Networks The team participates in the networks **Types** (type theory), see http://www.cs.chalmers.se/Cs/Research/Logic/Types/ and **Appsem 2** (Applied Semantics, concluded), see http://www.tcs.informatik.uni-muenchen.de/~mhofmann/appsem2/.

AlFA LERNET The team participates in the AlFA LERNET "LER-Language Engineering and Rigorous Software Development" project, a grant contract funded by the European Commission, started in March 2005, in which European and South American universities and research centres participate.

7.4. International projects and collaborations

STIC AmSud, Reseco project : Gilles Barthe is coordinator of the Reseco project within STIC AmSud, a regional program of scientific cooperation between France, Argentina, Brazil, Chile, Peru and Uruguay. The objective of the project is to investigate reliability and security in a computational model where both the platform and applications are dynamic, so that incoming software, built from off-the-shelf components, may be destined to form part of the platform or to execute as a standard application. The partners of the Reseco project are Oasis (INRIA-Sophia Antipolis), Chile University and Diego Portales University in Chile, Republic University in Montevideo, Uruguay and National University of Cordoba (FaMAF) in Argentina.

KTH, Stockholm, Sweden : we collaborate with D. Gurov and I. Aktug on the development of compositional verification techniques for control-flow security properties for (multi-threaded) applets.

8. Dissemination

8.1. Conference and workshop attendance, travel

- Tamara Rezk gave a presentation about her work at Intel, Portland, US, January 2006.
- Sabrina Tarento gave a presentation about her work at Verimag, Grenoble and in Caen, January 2006.
- Julien Charles and Marieke Huisman attended the meetings of the ACI Sécurité Geccoo in Grenoble, March 2006, and in Sophia Antipolis, June 2006. Julien Charles presented his work at the meeting in Grenoble.
- Gilles Barthe presented the paper [4] at FLOPS'06, Fuji Susono, JAPAN, April 2006.
- Benjamin Grégoire presented the paper [13] at FLOPS'06, Fuji Susono, JAPAN, April 2006.
- Gilles Barthe, David Pichardie and Tamara Rezk visited Chalmers, Sweden, for a Mobius meeting on type systems for information flow, and to collaborate with Andrei Sabelfeld and Alejandro Russo, April 2006.
- Mariela Pavlova attended Cardis'06, where one of her co-authors presented the paper [11], Tarragona, Spain, April 2006.
- Mariela Pavlova presented the paper [9] at SAC'06, Dijon, France, April 2006.
- Julien Charles visited Joseph Kiniry's team at UCD, April 2006
- Tamara Rezk presented the paper [7] at Security and Privacy'06, Oakland, US, May 2006.
- Tamara Rezk visited Stevens Institute in New Jersey, US, May 2006, to work with David Naumann.

- Tamara Rezk gave a presentation about her work at IBM Watson Research institute, New York, US, May 2006.
- Gilles Barthe, Julien Charles, Benjamin Grégoire, Sophie Hadjadj, Marieke Huisman, César Kunz, Mariela Pavlova, Gustavo Petri, David Pichardie, Tamara Rezk, and Santiago Zanella attended the annual Mobius meeting in Madrid, Spain, June 2006.
- Marieke Huisman presented the paper [14] at CSFW'06, Venice, Italy, July 2006. Gilles Barthe also attended the conference.
- Julien Charles presented the paper [10] at FTfJP'06, an ECOOP workshop, Nantes, France, July 2006.
- Gilles Barthe, Marieke Huisman and Tamara Rezk presented their work at the Dagstuhl seminar The Challenge of Software Verification, Germany, July 2006.
- Benjamin Grégoire presented the paper [12] at IJCAR'06, Seattle, August 2006.
- Tamara Rezk gave an invited talk at the PCC workshop at FLOC'06, August, 2006.
- Santiago Zanella presented the paper [16] at FAST'06, Hamilton, Canada, August 2006.
- Gilles Barthe presented the paper [6] at SAS'06, Seoul, Korea, August 2006.
- Clément Hurlin presented the paper [15] at AVOCS'06, Nancy, September 2006.
- Marieke Huisman and Mariela Pavlova were invited to present their work at the ESF workshop on Java Program Verification, Nijmegen, Netherlands, October 2006.
- Marieke Huisman presented a tutorial about Jack [21] at FMCO, Amsterdam, Netherlands, November 2006.
- Gilles Barthe presented the paper [5] at LPAR'06, Phnom Penh, Cambodia, November 2006.
- César Kunz gave an invited presentation about Certificate translation at ISoLA'06, Cyprus, November 2006.

8.2. Leadership within scientific community

- Gilles Barthe is scientific coordinator of:
  - the FET Integrated Project Mobius;
  - the ACI Security *SPOPS*;
  - the RNTL project *Castles*;
  - the Stic-Amsud project *Reseco*;

  and leader of Workpackage 2.6. Development Tools and Methodologies in the Integrated Project *Inspired*.
- Benjamin Grégoire and Marieke Huisman are task leaders for tasks in the Mobius project.
- Gilles Barthe was a member of the program committees for ESORICS'06, FM'06, LOPSTR'06, ISOLA'06, TGC'06, AMAST'06, SECRET'06, TFP'06, FAST'06.
- Marieke Huisman was a member of the program committees for .NET 2006, the VSTTE workshop at FLOC'06, the SAVCBS workshop at FSE'06.
- Gilles Barthe and Marieke Huisman were member of the jury for the Isabelle Attali Award (best innovative paper at E-smart 2006).
- Gilles Barthe is member of the Steering Committee of the Symposium on Trustworthy Global Computing.
- The team organised a meeting on the bytecode logic for the Mobius project in Sophia Antipolis, March 2006.

- The team organised a meeting about scenarios for Proof Carrying Code for the Mobius project in Sophia Antipolis, March 2006.
- With Laurent Théry (Marelle) and Benjamin Werner(LogiCal), Benjamin Grégoire organised a Workshop on Proofs & Numbers in Orsay, June 2006.
- The team organised a meeting about type systems for resource control for the Mobius project in Sophia Antipolis, November 2006.

8.3. Visiting scientists

We had several visiting scientists, many of whom gave a talk in our seminar. Dilian Gurov (KTH, Stockholm, Sweden) visited for three weeks in total, Lennart Beringer and Martin Hofmann (LMU, Munich) and Randy Pollack (U. Edinburgh, UK) visited for a week.

8.4. Supervision of Ph.D. projects

- Anne Pacalet is the local supervisor of Salvador Cavadini.
- Gilles Barthe supervised the Ph.D. projects of Mariela Pavlova (finished November 2006), Tamara Rezk (finished September 2006), Sabrina Tarento (finished September 2006) and supervises the Ph.D. project of Santiago Zanella (started June 2006).
- Benjamin Grégoire supervises the Ph.D. project of Julien Charles (started September 2005).
- Gilles Barthe and Benjamin Grégoire supervise the Ph.D. project of César Kunz (started October 2005).
- Marieke Huisman supervises the Ph.D. projects of Allard Kakebeen (resigned February 2006) and Clément Hurlin (started September 2006).

8.5. Ph.D. committees

- Gilles Barthe was a member of the Ph.D. jury's of Laurent Mazaré (Université de Grenoble, rapporteur), Nicolas Oury (Université Paris 11, rapporteur), Romain Janvier (Université de Grenoble, rapporteur), June Andronick (Université Paris 11), and Carlos Gonzalia (Göteborg University).

8.6. Supervision of internships

- Marieke Huisman supervised Gustavo Petri and Alejandro Tamalet.
- Gilles Barthe and Benjamin Grégoire co-supervised Fernando Pastawski.

8.7. Teaching

- Julien Charles taught Algorithms in Java, first year, and Unix, second year, University of Nice.
- Marieke Huisman taught: *Méthodes formelles* (Formal Methods), Ecole des Mines de Paris.
- Sabrina Tarento taught: *Programmation en C* (Programming in C), licence MP and Mass first year; *Programmation Java* (Programming in Java), licence MI first year; *Mathématiques intéractives sur internet* (Interactive mathematics on the internet), licence SM first year, University of Nice.

9. Bibliography

Year Publications

Books and Monographs

[1] G. Barthe, B. Grégoire, M. Huisman, J.-L. Lanet (editors). *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, Second International Workshop, CASSIS 2005, Nice, France, March 8-11, 2005, Revised Selected Papers*, Lecture Notes in Computer Science, vol. 3956, Springer, 2006.

Doctoral dissertations and Habilitation theses

[2] T. REZK. *Verification of confidentiality policies for mobile code*, Ph. D. Thesis, Université de Nice Sophia-Antipolis, 2006.

[3] S. TARENTO. *Formalisation en Coq de modeles cryptographiques et application au cryptosysteme ElGamal*, Ph. D. Thesis, Université de Nice Sophia-Antipolis, 2006.

Publications in Conferences and Workshops

[4] G. BARTHE, J. FOREST, D. PICHARDIE, V. RUSU. *Defining and reasoning about recursive functions: a practical tool for the Coq proof assistant*, in "Proceedings of FLOPS'06", Lecture Notes in Computer Science, vol. 3945, Springer-Verlag, 2006, p. 114-129, http://www-sop.inria.fr/everest/personnel/David.Pichardie/Publis/genfixpoint.pdf.

[5] G. BARTHE, B. GRÉGOIRE, F. PASTAWSKI. *Type-based termination of recursive definitions in the Calculus of Inductive Constructions*, in "Proceedings of the 13th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR'06)", LNAI, Springer-Verlag, November 2006, http://www-sop.inria.fr/everest/personnel/Benjamin.Gregoire/Publi/CICsombrero.pdf.gz.

[6] G. BARTHE, B. GRÉGOIRE, C. KUNZ, T. REZK. *Certificate Translation for Optimizing Compilers*, in "Proceedings of the 13th International Static Analysis Symposium (SAS), Seoul, Korea", LNCS, Springer-Verlag, August 2006, http://www-sop.inria.fr/everest/personnel/Cesar.Kunz/publications/certtrans-SAS06.pdf.

[7] G. BARTHE, D. NAUMANN, T. REZK. *Deriving an Information Flow Checker and Certifying Compiler for Java*, in "Proceedings of Symposium of Security and Privacy '06", IEEE Press, 2006.

[8] L. BURDY, M. HUISMAN, M. PAVLOVA. *Preliminary Design of BML: A Behavioral Interface Specification Language for Java bytecode*, in "Fundamental Approaches to Software Engineering (FASE 2007)", lncs, To appear., springer, 2007.

[9] L. BURDY, M. PAVLOVA. *Java Bytecode Specification and Verification*, in "SAC'06", ACM, 2006, http://www-sop.inria.fr/everest/personnel/Mariela.Pavlova/bcSpecVerify.pdf.

[10] J. CHARLES. *Adding Native Specifications to JML*, in "ECOOP workshop on Formal Techniques for Java-like Programs (FTfJP'2006)", 2006.

[11] A. COURBOT, M. PAVLOVA, G. GRIMAUD, J.-J. VANDEWALLE. *A Low-Footprint Java-to-Native Compilation Scheme Using Formal Methods*, in "CARDIS", 2006, p. 329-344.

[12] B. GRÉGOIRE, L. THERY. *A purely functional library for modular arithmetic and its application for certifying large prime numbers*, in "Proceedings of IJCAR'06", Lecture Notes in Artificial Intelligence, vol. 4130, Springer-Verlag, 2006, p. 423-437, http://www-sop.inria.fr/everest/personnel/Benjamin.Gregoire/Publi/numlib.pdf.

[13] B. GRÉGOIRE, L. THERY, B. WERNER. *A computational approach to Pocklington certificates in type theory*, in "Proceedings of FLOPS'06", Lecture Notes in Computer Science, vol. 3945, Springer-Verlag, 2006, p. 97 - 113, http://www-sop.inria.fr/everest/personnel/Benjamin.Gregoire/Publi/pock.pdf.

[14] M. Huisman, P. Worah, K. Sunesen. *A temporal logic characterisation of observational determinism*, in "19th IEEE Computer Security Foundations Workshop", IEEE Computer Society, July 2006, ftp://ftp-sop.inria.fr/everest/personnel/Marieke.Huisman/obsequiv_char.pdf.

[15] C. Hurlin. *Proof reconstruction for first-order logic and set-theoretical constructions*, in "Proc. International Workshop on Automated Verification of Critical Systems (AVOCS)", 2006.

[16] S. Zanella Béguelin, G. Betarte, C. Luna. *A Formal Specification of the MIDP 2.0 Security Model*, in "Proc. 4th International Workshop on Formal Aspects in Security and Trust, FAST 2006, Hamilton, Canada, August 26–27 2006", Lecture Notes in Computer Science, To appear, Springer-Verlag, 2006, http://www-sop.inria.fr/everest/personnel/Santiago.Zanella/MIDP/Zanella.2006.FAST.pdf.

Internal Reports

[17] G. Barthe, P. D'Argenio, T. Rezk. *Secure Information Flow by Self-Composition*, Technical report, INRIA, 2006, http://hal.inria.fr/inria-00097395.

[18] G. Barthe, D. Pichardie, T. Rezk. *Deriving an Information Flow Checker for the JVM*, Technical report, INRIA, 2006.

[19] D. Gurov, M. Huisman, C. Sprenger. *Compositional Verification of Sequential Programs with Procedures*, Technical report, INRIA, 2006.

Miscellaneous

[20] *Developer Oriented Methodology for Applet Validation*, Inspired Deliverable 9.1., 2006.

[21] G. Barthe, L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova, A. Requet. *JACK: a tool for validation of security and behaviour of Java applicationsFMCO*, Abstract for tutorial, longer version to appear, 2006.

References in notes

[22] A. Balaa, Y. Bertot. *Fix-point equations for well-founded recursion in type theory*, in "Proceedings of TPHOLs'00", M. Aagaard, J. Harrison (editors). , lncs, vol. 1689, springer, 2000.

[23] M. Bellare, P. Rogaway. *Code-Based Game-Playing Proofs and the Security of Triple Encryption*, 2004, http://eprint.iacr.org/, Cryptology ePrint Archive, Report 2004/331.

[24] J. Charles. *Vérification d'un composant Java: Le vérificateur de bytecode*, Technical report, Université de Nice, 2005, http://www-sop.inria.fr/everest/personnel/Julien.Charles/papers/05-06-17-rapport.pdf.

[25] S. Halevi. *A plausible approach to computer-aided cryptographic proofs*, 2005, http://eprint.iacr.org/, Cryptology ePrint Archive, Report 2005/181.

[26] J. Manson, W. Pugh, S. Adve. *The Java Memory Model*, in "to appear in ACM TOPLAS, Special POPL'05 Issue", 2005, http://www.cs.umd.edu/users/jmanson/java.html.

[27] M. PAVLOVA, G. BARTHE, L. BURDY, M. HUISMAN, J.-L. LANET. *Enforcing High-Level Security Properties For Applets*, in "Proceedings of CARDIS'04", P. PARADINAS, J.-J. QUISQUATER (editors). , kluwer, 2004, ftp://ftp-sop.inria.fr/everest/publis/P+04cardis.pdf.

[28] E. RODRÍGUEZ, M. B. DWYER, C. FLANAGAN, J. HATCLIFF, G. T. LEAVENS, ROBBY. *Extending JML for Modular Specification and Verification of Multi-Threaded Programs*, in "ECOOP 2005 — Object-Oriented Programming 19th European Conference, Glasgow, UK, Berlin", A. P. BLACK (editor). , Lecture Notes in Computer Science, vol. 3586, Springer–Verlag, July 2005, p. 551-576.

[29] V. SHOUP. *Sequences of games: a tool for taming complexity in security proofs*, 2004, http://eprint.iacr.org/, Cryptology ePrint Archive, Report 2004/332.

[30] S. TARENTO. *Machine-Checked Security Proofs of Cryptographic Signature Schemes*, in "Proceedings of ESORICS'05", S. DE CAPITANI DI VIMERCATI, P. SYVERSON, D. GOLLMANN (editors). , lncs, vol. 3679, springer, 2005, p. 140-158, http://www-sop.inria.fr/everest/Sabrina.Tarento/Papers/PA/main.pdf.