INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Project-Team EVEREST

Environnements de vérification et sécurité du logiciel

Sophia Antipolis

THEME SYM

Activity Report

2005

Table of contents

1. Team **1**
2. Overall Objectives **1**
2.1. Overall Objectives 1
3. Scientific Foundations **2**
3.1. Type systems 2
3.2. Program verification 2
3.3. Machine-checked semantics and algorithms 3
3.4. Software security 3
3.5. Proof Carrying Code 3
4. Application Domains **4**
4.1. Smart devices 4
4.2. Global computing 4
5. Software **5**
5.1. Jack: a tool for applet validation 5
5.2. Jakarta 5
6. New Results **5**
6.1. Program verification and Proof Carrying Code 5
6.2. Machine-checked semantics and algorithms 7
6.3. Type theory and proof assistants 7
7. Contracts and Grants with Industry **8**
7.1. Castles 8
8. Other Grants and Activities **8**
8.1. International collaborations 8
8.2. National initiatives 8
8.3. European initiatives 9
8.3.1. MOBIUS 9
8.3.2. INSPIRED 9
8.3.3. Thematic Networks 9
8.3.4. Other european projects 9
9. Dissemination **9**
9.1. Conference and workshop attendance, travel 9
9.2. Leadership within scientific community 10
9.3. Visiting scientists 10
9.4. Supervision of Ph.D. projects 10
9.5. Ph.D. committees 10
9.6. Supervision of internships 11
9.7. Teaching 11
10. Bibliography **11**

1. Team

Team Leader
Gilles Barthe [Research Director INRIA]

Team Vice-Leader
Marieke Huisman [Research scientist INRIA]

Research scientist
Benjamin Grégoire [INRIA, since September 2005]

Administrative assistant
Nathalie Bellesso

Scientific advisor
Jean-Louis Lanet [INRIA DirDRI, until March 2005, now at Gemplus]

Ph. D. student
Julien Charles [since october 2005, MESR grant, Teaching Assistant UNSA]
Allard Kakebeen [since november 2005]
César Kunz [from october 2005]
Mariela Pavlova
Tamara Rezk
Sabrina Tarento [MESR grant, Teaching Assistant UNSA]

Post-doctoral Fellow
Julien Forest [since February 2005]
Florian Kammüller [until march 2005]

Technical Staff
Benjamin Grégoire [until september 2005, now researcher at INRIA]
David Pichardie [since september 2005]

Ph.D. visiting students
Luca Martini [PhD student at University of Pisa, 3 months]

Student internship
Julien Charles [Nice-Sophia Antipolis University, Master internship, 7 months]
Maoulida Daoudou [Provence University, Master internship, 6 months]
Thibault Dupont [Nice-Sophia Antipolis University, Master internship, 6 months]
César Kunz [Cordoba University, Internship Program 4 months]
Jorge Luis Sacchini [Rosario University, Internship Program, 4 months]
Dante Zanarini [Rosario University, Internship Program, 4 months]

2. Overall Objectives

2.1. Overall Objectives

The Everest project aims at ensuring security for mobile and embedded applications. Our privileged application domains are small, trusted and portable devices, such as mobile phones and smart cards, their operating systems and the applications running on these devices.

In a more general sense, our work concentrates on the development and validation of secure software, both at platform and application level. To achieve this, we advocate the use of formal methods and language-based techniques for safe and secure programming.

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, 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 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 exist for describing computational systems.

- *Logic is precise*. While types generally over-approximate program behavior, logic can be used to provide precise statements of program behavior. Special conditions can be assumed and exceptional behaviors can be described. Via the use of negation and rich quantifier alternation, it is possible to state nearly arbitrary observations about programs and computational systems.
- *Logic allows to combine analyses*. Logic provides a common setting into which the declarative content of a typing judgment or other static analyses can be translated. The results of such analyses can then be placed into a common logic so that conclusions about their combinations can be drawn.

Recently, there has been a lot of research into program verification of Java, which has culminated in the realization 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 modeling 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 smartcards, 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 formalized 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 realize 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 smartcards 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 organizations 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 card 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 base of software (an abstract 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 smart devices (so-called post issuance), and by allowing several commercially independent applications to run on a single smart card. This open character forms its commercial strength, but also creates the technical challenges of these devices: reliability, correctness and security become crucial issues, since malicious applets might potentially exploit bugs in the smart card 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 unauthorized 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, Lilian Burdy, Julien Charles, Benjamin Grégoire, Marieke Huisman, Jean-Louis Lanet, Mariela Pavlova.

We develop Jack, the Java Applet Correctness Kit, a tool for applet validation. Jack takes Java or JVM programs annotated with JML specifications (or JML-like specifications for JVM programs) as input, and generates appropriate proof obligations. These can be used as input for different proof assistants (automatic or interactive), such as Simplify and Coq.

Jack is integrated in Eclipse, and it features a user friendly interface to study the proof obligations, including colouring of the relevant code and different views for the proof obligations. Moreover, the tool can generate assertions to encode high-level security properties that can be described as finite state machines.

In addition, the tool includes a compiler that takes as input a Java applet annotated with JML annotations, and generates the corresponding bytecode program and JML-like annotations for the bytecode. The next step is to provide an explicit representation of proofs, a.k.a. proof objects, and to extend the compiler to also work on these proof objects, in order to offer support for Proof Carrying Code.

5.2. Jakarta

Participants: Gilles Barthe, Guillaume Dufay, Julien Forest.

We develop a tool for specifying and verifying formally 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 JavaCard bytecode verifier.

The development of the tool was initiated in the context of the european project Verificard and is partly done in the context of the RNTL project CASTLES which focuses on the certification of execution platforms for smartcards.

6. New Results

6.1. Program verification and Proof Carrying Code

Participants: Gilles Barthe, Lilian Burdy, Julien Charles, Benjamin Grégoire, Marieke Huisman, Allard Kakebeen, César Kunz, Jean-Louis Lanet, Mariela Pavlova, Tamara Rezk, Dante Zanarini.

1. We studied certificate translators for optimizing compilers, and in particular the means to transform certificates of source code into certificates of executable code in the context of an optimizing compiler [19]. 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 also studied the derivation of an Information Flow Checker and Certifying Compiler for Java. This work describes a systematic technique to connect source code and bytecode security type systems. The technique is applied to an information flow type system for a fragment of Java with exceptions, thus confronting challenges in both control and data flow tracking.
3. Several new features have been added 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 [20]. Together with the Coq plugin, we also developed several tactics to solve or simplify some of Jack's proof obligations automatically (and therewith increase their readability). To avoid too large computation times, this has been done in a light-weight fashion. We have developed two kinds of tactics: one to clean up goals by removing useless information, and one to solve goals using of Jack's specific constructs.
     Note that a similar Coq plug-in has been developed for ESC/Java.
   – A plugin (JML2BC) to translate JML specifications into bytecode specifications, and a plugin (BcVcGen) to generate verification conditions from Java bytecode and its specification [13].
     Both JML2BC and BcVcGen could be used in mobile code scenarios. In particular, the JML2BC plugin allows that the bytecode is shipped with its specification at the client site (thus making it unnecessary to infer specifications, which would severely limit the kind of properties that can be expressed). The BcVcGen can be used on the client site to generate proof obligations, that ensure that the implementation satisfies the specification - thus establishing trust in the unknown code.

4. To validate the Jack tool we have annotated a Java bytecode verifier [20]. 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 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.
5. We studied memory consumption policies using logical methods [8]. We propose a new methodology to verify that a Java bytecode program respects some given memory consumption constraint using Hoare style logic. The methodology requires that one specifies method pre- and postconditions and loop invariants to express that no method will use more memory that the whole application is allowed to. Part of this specification is inferred automatically (for example an upper bound of the memory consumed by a method), and the other part must be supplied by the code producer (for example an upper bound on the number of loop iterations for every loop). The memory consumption is then verified by generating appropriate verification conditions from the annotated bytecode.
6. The work on compositional verification is currently being extended to multi-threading and exceptions. For multi-threading, the compositional verification principle carries over immediately, but the verification problem becomes undecidable. For exceptions, a new notion of applet simulation will have to be defined.
7. In collaboration with the university of Besançon, we study means to generate appropriate JML annotations to express liveness properties. If one can show that the JML annotations are verified, and that the environment is a refinement of the ideal environment (*i.e.* it calls the relevant methods sufficiently often), then one can conclude that the liveness property is satisfied.

6.2. Machine-checked semantics and algorithms

Participants: Gilles Barthe, Julien Forest, Benjamin Grégoire, Marieke Huisman, Allard Kakebeen, Maoulida Daoudou, David Pichardie, 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. The current version of the Jakarta tool can only abstract a defensive machine into both an offensive one (which does not perform any security tests) and a typed one (which only performs security tests) and proves the correction of the abstraction. It would be interesting to be able to revert the process and to obtain a defensive machine from both an offensive and a typed one (in particular if we want to merge different security tests). We have developed in Coq a first attempt to obtain the defensive machine from the the offensive and the typed machine [21]. This uses Ltac, the tactic language of Coq.
3. We have developed a formal semantics in Coq of a representative fragment of the Java Bytecode language, using the Coq proof assistant. The name of the project is Bicolano (Byte Code Language in cOq). This work is done in collaboration with the Lande team at IRISA. This semantics will be used in the Mobius project as a reference for the forthcoming formal developments on Java Bytecode. The semantics is available from the team web page.
4. We have pursued our work on the formalization 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 [12], we have shown an upper bound to the probability of an interactive adversary (that adheres to the GM and to the ROM i.e., he makes some interactions with a hash oracle and a decryption oracle) to find a certain secret. In [18], we have also considered the case of parallel attacks, we have shown an upper bound to the probability of an interactive adversary (that adheres to the GM and to the ROM i.e., he makes some interactions with a hash oracle and a signature oracle) to make an one-more signature forgery.

6.3. Type theory and proof assistants

Participants: Gilles Barthe, Julien Forest, Benjamin Grégoire, David Pichardie, Jorge Luis Sacchini.

1. We have developed a practical method to define and reason about general recursive functions in Coq [6]. This method allows the user to easily define and reason about not-structurally recursive functions in Coq. A tool has been implemented in Ocaml and in Coq. The tool proceeds by generating from pseudo-code (Coq functions that need not be neither total nor terminating) the *graph* of the intended function as an inductive relation, and then proves that the relation actually represents a function. The proof proceeds by constructing the function that we are trying to define. Then, the tool generates *induction and inversion principles*, and a *fixpoint equation* for proving other properties of the function. The tool builds upon state-of-the-art techniques for defining recursive functions, and can also be used to generate executable functions from inductive descriptions of their graph.
2. We have studied how to add exceptions in dependent type theory [22]. We have first defined a λ-calculus with data types and exceptions that allows strong reduction and confluence (these two properties are needed to show logical consistency of the type theory). Then we have tried to extend our calculus with dependent types, unfortunately this leads to a system that is too complicated to be useful.

3. In proof systems like Coq, proof-checking involves comparing types modulo β-conversion. To speed up this conversion test, Grégoire and Leroy use a compiled approach that erases most of the type decorations carried by the proofs (domains of abstractions and parametric arguments of constructors). We have shown that erasure does not change the underlying formalism for a large class of systems including the Calculus of Inductive Constructions (Coq's formalism) and Cumulative Type Systems [4].

7. Contracts and Grants with Industry

7.1. Castles

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/. Goal of the project is to define an environment to support the formal specification and verification of low-level execution platforms.

8. Other Grants and Activities

8.1. International collaborations

- KTH, Stockholm, Sweden: compositional verification of control-flow security properties for multi-threaded applets.
- Stevens Institute of Technology, and Kansas State University, USA: language-based security.
- GlobalPlatform: formal models of the GlobalPlatform specifications.

8.2. National initiatives

- ACI Sécurité GECCOO (Generation of Certified Code for Object-Oriented Applications - Specifications, refinement, proof and error detection, 2003 - 2006). Other participants are TFC (Besançon), Cassis (Nancy), LogiCal (LRI/Futurs) and Vasco (IMAG, Grenoble). For more information, see http://geccoo.lri.fr/.
- ACI Sécurité SPOPS (Secure Operating Systems for Trusted Personal Devices, 2003 - 2006). Other participants are POPS (Lille) and SSIR (Supélec, Rennes). For more information, see http://www-sop.inria.fr/everest/projects/spops/.

8.3. European initiatives

8.3.1. MOBIUS

We are 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 (September 2005 - September 2009). The project gets together 12 academic and 4 industrial partners. Goal of the project is to develop the technology for establishing trust and security for the next generation of global computers, using the Proof Carrying Code (PCC) paradigm. The essential features of the Mobius security architecture will be innovative trust management, static enforcement mechanisms and support for system component downloading. For more information, see http://mobius.inria.fr.

8.3.2. INSPIRED

We participate in the European IST project Inspired (December 2003 - December 2006). The partners in this 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. For more information, see http://www.inspiredproject.com/.

8.3.3. Thematic Networks

EVEREST participates in the network Types (type theory) (see http://www.cs.chalmers.se/Cs/Research/Logic/Types/ and Appsem (Applied Semantics) (see http://www.tcs.informatik.uni-muenchen.de/~mhofmann/appsem2/).

8.3.4. Other european projects

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

9. Dissemination

9.1. Conference and workshop attendance, travel

- Gilles Barthe attended the FET-ERCIM Workshop on "Security, Dependability and Trust", Brussels, Belgium, on October 12, 2005
- Gilles Barthe was invited to attend the French-Japonese symposium in Tokyo, Japan, 5-7 September 2005.
- Gilles Barthe presented his work at TLCA, Nara, Japan, 21-23 April 2005.
- Gilles Barthe attended ETAPS, Edinburgh, United Kingdom, 2-10 April 2005.
- Gilles Barthe was invited to attend the kick-off seminar of STIC AmSud at Santiago, Chili, 19-20 December 2005.
- Julien Charles presented his work at a meeting of the ACI Sécurité Geccoo in March 2005. Together with Benjamin Grégoire, he also attended a Geccoo meeting in October 2005.
- Julien Forest attended the Second Workshop on the Rho Calculus, at LIX, Polytechnique School, Palaiseau in May 2005.
- Julien Forest presented his work at the seminar of Lami, Evry, in April 2005 and at the seminar of the Lande Project, IRISA, Rennes in April 2005.
- Marieke Huisman visited KTH, Stockholm, Sweden, 12-16 December 2005.
- Mariela Pavlova visited the Computer science Laboratory of Lille University, June 2005.

- Mariela Pavlova presented her work at SEFM'05, Koblenz, Germany, 4-9 September 2005.
- Mariela Pavlova attended the Summer School ARTIST2, Nässlingen, Sweden, 29 September until 3 October 2005.
- Tamara Rezk presented her work at TLDI'05, January 10, 2005 and at POPL, Long Beach, USA, 12-14 January 2005.
- Tamara Rezk attended the VSSTE Workshop, Zurich, Switzerland, 10-13 October 2005.
- Tamara Rezk visited Anindya Banerjee, Kansas State University, USA, in January 2005 to work on non-interference for software with pointer programs.
- Sabrina Tarento presented her work at the LORIA Workshop on the link between formal and computational models, Paris, June 2005, and at ESORICS05, September 2005.

9.2. Leadership within scientific community

- Gilles Barthe, Marieke Huisman, Jean-Louis Lanet and Benjamin Grégoire co-organized the International Workshop on Construction and Analysis of Safe, Secure and Interoperatble Smart Devices (CASSIS), Nice, March 2005. The proceedings will appear as a volume of Lecture Notes in Computer Science.
- Marieke Huisman was a member of the program committee for the ETAPS workshop Bytecode'05.
- Marieke Huisman was a member of the jury for the Isabelle Attali Award (best innovative paper at E-smart 2005).
- Gilles Barthe was a member of the programme committee for APPSEM'05, FCS'05, LOPSTR'05,WWV'05.
- Benjamin Grégoire gave an invited lecture on *Dependently Typed programming*, TYPES Summer School 2005, August 15-26 2005, Göteborg, Sweden.
- Gilles Barthe gave an invited lecture on *Méthodes formelles de sécurité logicielle* (Formal methods for software security), Ecole Jeunes Chercheurs en Programmation 2005, Dinard; *Formal methods for software security*, Rio Cuarto Summer School, Argentina; *Dependent Type Theory* APPSEM II Summer School, Munich, Germany ; *Formal methods for smartcards* Fosad 2005 Summer School, Bertinoro, Italy.

9.3. Visiting scientists

We had several visiting scientists, many of whom gave a talk in our seminar. Dilian Gurov and Irem Aktug (KTH, Stockholm, Sweden) both visited for a week.

9.4. Supervision of Ph.D. projects

- Gilles Barthe supervises the Ph.D. project of Mariela Pavlova, Tamara Rezk, Sabrina Tarento.
- Benjamin Grégoire supervises the Ph.D. project of Julien Charles (started October 2005).
- Gilles Barthe and Benjamin Grégoire supervise the Ph.D. project of César Kunz (started October 2005).
- Marieke Huisman supervises the Ph.D. project of Allard Kakebeen (started November 2005).

9.5. Ph.D. committees

- Gilles Barthe was rapporteur for the Habilitation of Claude Marché.

9.6. Supervision of internships

- Benjamin Grégoire supervised Julien Charles.
- Gilles Barthe and Julien Forest co-supervised Maoulida Daoudou.
- Gilles Barthe supervised Thibault Dupont and Dante Zanarini.
- Gilles Barthe and Benjamin Grégoire co-supervised César Kunz and Jorge Luis Sacchini.

9.7. Teaching

- Benjamin Grégoire and Gilles Barthe and David Pichardie taught: *Semantics, Security and Verification*, UNSA, October-December 2005.
- Jean-Louis Lanet taught: *Sécurité du logiciel* (Software security), DESS TIR, Université de Lille.
- Mariela Pavlova taught: *Informatique Théorique* (Theoretical Computer Science), Licence MASS2, University of Nice.
- Sabrina Tarento taught: *Programmation en C* (Programming in C), licence MP first year; *Programmation Java* (Programming in Java), licence MI first year; *Mathématiques interactives sur Internet*(Interactive mathematics on the internet), licence SM first year; *Calcul Formel* (Formal calculus); *Algèbre* (Computer Algebra) and *Introduction à la programmation en C* (Introduction to programming in C) licence MASS 1, University of Nice.

10. Bibliography

Articles in refereed journals and book chapters

[1] G. BARTHE. *Type Isomorphisms and Back-and-Forth Coercions in Type Theory*, in "Mathematical Structures in Computer Science", To appear, 2005.

[2] G. BARTHE, T. REZK, A. BASU. *Security Types Preserving Compilation*, in "Journal of Computer Languages, Systems and Structures", To appear, 2005.

[3] C. BREUNESSE, N. CATAÑO, M. HUISMAN, B. JACOBS. *Formal Methods for Smart Cards: an experience report*, in "Science of Computer Programming", vol. 55, n° 1-3, 2005, p. 53-80.

Publications in Conferences and Workshops

[4] B. BARRAS, B. GRÉGOIRE. *On the role of type decorations in the Calculus of Inductive Constructions*, in "Proceedings of CSL'05, Oxford, UK", to appear, August 2005, http://www-sop.inria.fr/everest/personnel/Benjamin.Gregoire/Publi/equiv.ps.gz.

[5] G. BARTHE, G. DUFAY. *Formal methods for smartcard security*, in "Proceedings of FOSAD'05", A. ALDINI, R. GORRIERI, F. MARTINELLI (editors). , Lecture Notes in Computer Science, vol. 3655, Springer-Verlag, 2005, p. 133–177.

[6] 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", LNCS, To appear, 2006.

[7] G. BARTHE, B. GRÉGOIRE, F. PASTAWSKI. *To Practical inference for typed-based termination in a polymorphic setting*, in "Proceedings of TLCA'05, Nara, Japan", P. URZYCZYN (editor). , Lecture Notes in Computer Science, vol. 3641, Springer-Verlag, April 2005, p. 71-85.

[8] G. BARTHE, M. PAVLOVA, G. SCHNEIDER. *Precise analysis of memory consumption using program logics*, in "Proceedings of SEFM'05, Koblenz, Germany", B. AICHERNIG, B. BECKERT (editors). , IEEE Press, September 2005, p. 86–95.

[9] G. BARTHE, T. REZK. *Non-interference for a JVM-like language*, in "Proceedings of TLDI'05, Long Beach, USA", M. FÄHNDRICH (editor). , ACM Press, January 2005, p. 103–112.

[10] G. BARTHE, T. REZK, A. SAABAS. *Proof obligations preserving compilation*, in "Proceedings of FAST'05", R. GORRIERI, F. MARTINELLI, P. RYAN, S. SCHNEIDER (editors). , Lecture Notes in Computer Science, To appear, Springer-Verlag, 2005.

[11] G. BARTHE, T. REZK, M. WARNIER. *Preventing Timing Leaks Through Transactional Branching Instructions*, in "Proceedings of 3rd Workshop on Quantitative Aspects of Programming Languages (QAPL'05), Edinburgh, Scotland", to appear, Electronic Notes in Theoretical Computer Science, 2005.

[12] G. BARTHE, S. TARENTO. *A Machine-Checked Formalization of the Random Oracle Model*, in "Proceedings of TYPES'04", Lecture Notes in Computer Science, To appear, Springer-Verlag, 2005.

[13] L. BURDY, M. PAVLOVA. *Java Bytecode Specification and Verification*, in "Proceedings of SAC'06", ACM, to appear, 2006.

[14] B. GRÉGOIRE, Y. BERTOT, X. LEROY. *A structured approach to proving compiler optimizations based on dataflow analysis*, in "Proceedings of TYPES'04", Lecture Notes in Computer Science, To appear, Springer-Verlag, 2005, http://www-sop.inria.fr/everest/personnel/Benjamin.Gregoire/Publi/concert.ps.gz.

[15] B. GRÉGOIRE, A. MAHBOUBI. *Proving equalities in a commutative ring done right in Coq*, in "Proceedings of TPHOLs'05, Oxford, UK", J. HURD, T. MELHAM (editors). , Lecture Notes in Computer Science, vol. 3603, Springer, August 2005, p. 98–113, http://www-sop.inria.fr/everest/personnel/Benjamin.Gregoire/Publi/newring.ps.gz.

[16] D. GUROV, M. HUISMAN. *Interface Abstraction for Compositional Verification*, in "Proceedings of SEFM'05, Koblenz, Germany", B. AICHERNIG, B. BECKERT (editors). , IEEE Press, September 2005, p. 414-423.

[17] M. HUISMAN, K. TRENTELMAN. *Factorising temporal specifications*, in "Proceedings of CATS'05, Newcastle, Australia", M. ATKINSON, F. DEHNE (editors). , Conferences in Research and Practice in Information Technology, An earlier version appeared as INRIA Technical Report, nr. RR-5326, vol. 41, ACSC, February 2005, p. 87–96.

[18] 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). , Lecture Notes in Computer Science, vol. 3679, Springer-Verlag, 2005, p. 140-158.

Miscellaneous

[19] G. BARTHE, B. GRÉGOIRE, C. KUNZ, T. REZK. *Certificate translation for optimizing compilers*, Manuscript, 2005.

[20] J. CHARLES. *Vérification d'un composant Java: le vérificateur de bytecode*, Technical report, Université de Nice Sophia-Antipolis, 2005.

[21] M. DAOUDOU. *From defensive semantic to offensive and typed ones using LTac*, Technical report, Université de Marseille, 2005.

[22] J. SACCHINI. *Exceptions in dependent type theory*, Internship report, 2005.