Computer Security from a Programming Language and Static Analysis Perspective
(Extended Abstract of Invited Lecture)

Xavier Leroy

INRIA Rocquencourt and Trusted Logic S.A.
Domaine de Voluceau, B.P. 105, 78153 Le Chesnay, France
Xavier.Leroy@inria.fr

1 Introduction

Computer security [16,5] is usually defined as ensuring integrity, confidentiality, and availability requirements even in the presence of a determined, malicious opponent. Sensitive data must be modified and consulted by authorized users only (integrity, confidentiality); moreover, the system should resist "denial of service" attacks that attempt to render it unusable (availability). In more colorful language, computer security has been described as "programming Satan's computer" [6]: the implementor must assume that every weakness that can be exploited will be.

Security is a property of a complete system, and involves many different topics, both computer-related (hardware, systems, networks, programming, cryptography) and user-related (organizational and social policies and laws). In this talk, we discuss the impact of programming languages and static program analysis on the implementation of access control security policies, with special emphasis on smart cards. By lack of time, we will not discuss other relevant examples of programming language concepts being used for computer security, such as type systems for information flow [42,41,20,2,34,35] and validation of cryptographic protocols using process algebras and types [4,1,3].

2 Access Control

Access control is the most basic and widespread security policy. An access control policy provides yes/no answers to the question "is this principal (user, program, role, ...) allowed to perform this operation (read, write, creation, deletion, ...) on this resource (file, network connection, database record, ...)?". Access control is effective to ensure integrity, and can also ensure simple confidentiality properties.

2.1 Preventing Direct Access to a Resource

Access control is performed by fragments of code (OS kernel, reference monitor, privileged library) that check that access to a logical resource (file, network

P. Degano (Ed.): ESOP 2003, LNCS 2618, pp. 1–9, 2003.

connection) is allowed before performing the operation on the underlying low-level resource (disk or network controller). Of course, access control is moot if the program can bypass this code and operate directly on the low-level resource.

The traditional answer to this issue relies on hardware-enforced mechanisms: the low-level resources can only be accessed while the processor is in supervisor mode, and switching from user mode to supervisor mode can only be performed through specific entry points that branch to the access control code. On the user side, resources are represented indirectly by "handles", e.g. indices into kernel tables. Hardware memory management prevents user code from accessing kernel data directly.

This model, while effective, is not always suitable. Sometimes, user-mode programs must be further partitioned into relatively trusted (Web browser) and completely untrusted (Web applets). Switching between user and supervisor modes can be expensive. The required hardware support may not be present, e.g. in small embedded devices.

An alternate, language-based approach executes all code within the same memory space, without hardware protections, but relies on strong typing to restrict direct access to sensitive resources. These resources are directly represented by pointers, but strong typing prevents these pointers from being forged, e.g. by guessing their addresses. Thus, the typing discipline of the language can be used to enforce security invariants on the resources.[1]

As a trivial example, if a resource is not reachable from the initial memory roots of a piece of code, memory safety, also called garbage collection safety, ensures that this code can never access this resource. As a less trivial example, two standard type-based encapsulation techniques can be used to provide controlled access to a resource: procedural encapsulation and type abstraction [27].

- With procedural encapsulation, the resource is a free variable of a function closure, or a private field of an object, and only the closure or the object are given to the untrusted code. The latter, then, cannot fetch the resource pointers directly from the object or the closure (this would be ill-typed), and must call the function or a method of the object to operate on the resource; the code of the function or the method will then perform the required access checks before performing the operation.
- With type abstraction, the resource pointer itself can be given to the untrusted code, but its type is made abstract, preventing the code from operating directly on it; to use the resource, the code must call one of the operations provided in the signature of the abstract type, and this code will then perform access checks as described before.

As outlined above, strong typing can be exploited to enforce access control. The remaining question, then, is how to enforce a strong typing discipline during execution of (untrusted) code. A simple approach is to perform type checks dynamically, during program execution. This can be achieved in many ways: direct

[1] Strong typing is also effective at preventing other kinds of attacks such as buffer overflows that cause attacker-provided data to be executed as privileged code.

interpretation of the program source (if available); compilation of the source with insertion of run-time checks; bytecode interpretation of virtual machine code such as the JVM [28]; just-in-time compilation of said virtual machine code; and instrumentation of precompiled machine code with additional checks (software fault isolation) [43].

To reduce the run-time overhead of dynamic type checking, it is desirable to perform some of the type checks statically, during a program analysis pass prior to actual execution. Static typing of source code is common and well understood [8]. However, source for untrusted code is not always available. Moreover, bugs in the source-to-executable compiler could introduce type violations after type checking; in other terms, the compiler is part of the trusted computing base. These drawbacks can be avoided by performing static type-checking on lower-level, compiled code. A famous example is Java bytecode verification [18,28, 25], which performs static type-checking on JVM bytecode at dynamic loading time. Typed Assembly Language [31,30] goes several steps below virtual machine code: it statically type-checks assembly code for an actual microprocessor (the Intel $x86$ family), including many advanced idioms such as stack unwinding for exception handling.

Java bytecode verification and typed assembly language leave certain checks relevant to type safety to be performed at run-time: typically, array bound checks, or Java's downcasts. More advanced type systems based on dependent types were proposed to allow static verification of array bound checks (and more) [47,46,38,13]. Proof-carrying code [32] takes this approach to the extreme by replacing static type checking with static proof checking in a general program logic: the provider of the code provides not only compiled code, but also a proof that it satisfies a certain security property; the user of the code, then, checks this proof to make sure that the code meets the property. The property typically includes type correctness and memory safety, but can also capture finer behavioral aspects of the code [33].

2.2 Implementing Access Control

The security policy implemented by access control checks is traditionally represented as an access control matrix, giving for each resource and each principal the allowed operations. This matrix is often represented as access control lists (each resource carries information on which principals can access it) or as capabilities (each principal carries a set of resources that it can access). The yes/no nature of access control matrices is sometimes too coarse: security automata [37] can be used instead to base access control decisions on the history of the program execution, e.g. to allow an applet to read local files or make network connections, but not both (to prevent information leaks).

Determining the principal that is about to perform a sensitive operation is often difficult. In particular, shared library code that performs operations on behalf of an untrusted user must have lower privileges than when performing operations on behalf of a trusted user. The Java security manager [17] uses stack inspection to address this problem: each method is associated with a principal,

and the permissions granted are those of the least privileged principal appearing in the method call stack leading to the current operation. This model is sometimes too restrictive: an applet is typically allowed to draw text on the screen, but not to read files; yet, to draw text on behalf of the applet, the system may need to read fonts from files. Privilege amplification mechanisms are provided to address this need, whereas system code can assert a permission (e.g. read a file for the font loading code) regardless of whether its caller has it.

Access control checks are traditionally performed dynamically, during execution. The run-time overhead of these checks is generally tolerable, and can be further reduced by partial evaluation techniques allowing for instance inline expansion and specialization of security automata [15,12,40].

Still, it is desirable to perform static approximations of access control checks: to guide and validate optimizations such as removal of redundant checks, but also to help programmers determine whether their code works correctly under a given security policy. Jensen *et al.* [7] develop a static approximation of the Java stack inspection mechanism, where the (infinitely many) call stacks are abstracted as a finite automaton, and security properties described as temporal formulae are model-checked against this automaton. Pottier *et al.* [36] compose the security-passing interpretation of stack inspection (proposed in [45] as a dynamic implementation technique) with conventional type systems described in the $HM(X)$ framework to obtain type-based static security analyses. Finally, Walker [44] describes a type system for typed assembly language where the states of security automata are expressed within the types themselves, allowing fine static control of the program security behavior.

3 Application to Smart Card Programming

3.1 Smart Card Architectures

Smart cards are small, inexpensive embedded computers used as security tokens in several areas, such as credit cards and mobile phones. Traditional smart cards such as Eurocard-Mastercard-Visa credit cards behave very much like a small file system, with access control on directories and files, and determination of principals via PIN codes.

The newer Java Card architecture [10] offers a model closer to an applet-enabled Web browser, with several applications running in the same memory space, and post-issuance downloading of new applications. The applications are executed by a virtual machine that is a subset of the JVM. The security of this architecture relies heavily on the type safety of this JVM variant. For access control, the Java security manager based on stack inspection is replaced by a simpler "firewall" that associates owners to Java objects and prevents an application from accessing directly an object owned by another application. Inter-application communications are restricted to invocation of interface methods on objects explicitly declared as "shared".

Formal methods are being systematically applied to many aspects of the Java Card architecture [19]: formal specifications of the virtual machine, applet

loading, the firewall, the APIs, and specific applications; and machine proofs of safety properties such as type safety and non-interference. As for program analyses, several approaches to on-card bytecode verification have been proposed [26,14]. Static analyses of firewall access control are described in [9]. Chugunov *et al.* [11] describe a more general framework for verifying safety properties of Java Card applets by model checking.

3.2 Hardware Attacks

The software-based security techniques presented in section 2 all assume that the programs execute on reliable, secured hardware: the best software access control will not prevent information leaks if the attacker can simply steal the hard disk containing the confidential data. In practice, hardware security is often ensured by putting the computers in secured premises (locked doors, armed guards).

For smart cards and similar embedded devices, this is not possible: the hardware is physically in the hands of the potential attackers. By construction, a smart card cannot be pulled apart as easily as a PC: smart card hardware is designed to be tamper proof to some extent. Yet, the small size and cost of a smart card does not allow extensive tamper proofing of the kind used for hardware security modules [39]. Thus, a determined attacker equipped with a good microelectronics laboratory can mount a variety of physical attacks on a smart card [23]:

- Non-intrusive observation: perform precise timings of operations; measure power consumption or electromagnetic emissions as a function of time.
- Intrusive observation: expose the chip and implant micro-electrodes on some data paths.
- Temporary perturbation: introduce "glitches" in the power supply or the external clock signal; "flash" the chip with high-energy particles.
- Permanent modification: destroy connections and transistors within the chip.

These attacks can defeat the security of the software in several ways. Power analysis can reveal the sequencing of instructions performed, thus reveal secret data such as the private keys in naive implementation of public-key cryptography [22]. Perturbations or modifications can prevent some instructions of the program from executing normally: for instance, a taken conditional branch can be skipped, thus deactivating a security check. Variables and registers can also be set to incorrect values, causing for instance a loop intended to send a communication buffer on the serial port to send a complete memory dump instead.

3.3 Software Countermeasures

The obvious countermeasure to these attacks is to harden the smart card hardware [24]. It is a little known fact that the programs running on smart cards can also be written in ways that complicate hardware attacks. This is surprising, because in general it is nearly impossible for a program to protect itself from

execution on malicious hardware. (Some cryptographic techniques such as those described in [29] address this issue in the context of boolean circuits, but have not been demonstrated to be practical.)

The key to making smart card software more resistant is to notice that hardware attacks cannot change the behavior of the hardware arbitrarily. Permanent modifications are precise but irreversible, thus can be detected from within the program by running frequent self tests, and storing data in a redundant fashion (checksums). Temporary perturbations, on the other hand, are reversible but imprecise: they may cause all the memory to read as all zeroes or all ones for a few milliseconds, but cannot set a particular memory location to a particular value. Thus, their impact can be minimized by data redundancy, and also by control redundancy. For instance, a critical loop can be double-counted, with one counter that increases and another that decreases to zero; execution is aborted if the sum of the two counters is not the expected constant.

Finally, hardware attacks can be made much harder by program randomization. Randomizing data (as in the "blinding" technique for RSA [22]) renders information gained by power analysis meaningless. Randomizing control (e.g. calling independent subroutines in a random order, or choosing randomly between different implementations of the same function) makes it difficult to perform a perturbation at a given point in the program execution.

Software hardening techniques such as the ones outlined above are currently applied by hand on the source code, and often require assembly programming to get sufficient control on the execution. It is interesting to speculate how modern programming techniques could be used to alleviate this burden. The hardening code could possibly be separated from the main, algorithmic code using aspect-oriented programming [21]. Perhaps some of the hardening techniques are systematic enough to be performed transparently by a compiler, or by a virtual machine interpreter in the case of Java Card. Finally, reasoning about software hardening techniques could require a probabilistic semantics that reflects some of the time-precision characteristics of likely hardware attacks.

References

1. Martín Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, 1999.
2. Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *26th symposium Principles of Programming Languages*, pages 147–160. ACM Press, 1999.
3. Martín Abadi and Bruno Blanchet. Secrecy types for asymmetric communication. In Furio Honsell and Marino Miculan, editors, *Proceedings of the 4th International Conference on Foundations of Software Science and Computation Structures*, volume 2030 of *Lecture Notes in Computer Science*, pages 25–41. Springer-Verlag, 2001.
4. Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The Spi calculus. *Information and Computation*, 148(1):1–70, 1999.
5. Ross Anderson. *Security Engineering.* John Wiley & Sons, 2001.

6. Ross Anderson and Roger Needham. Programming Satan's computer. In *Computer Science Today: Recent Trends and Developments*, number 1000 in Lecture Notes in Computer Science, pages 426–441. Springer-Verlag, 1995.
7. Frédéric Besson, Thomas de Grenier de Latour, and Thomas Jensen. Secure calling contexts for stack inspection. In *Principles and Practice of Declarative Programming (PPDP 2002)*, pages 76–87. ACM Press, 2002.
8. Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*. CRC Press, 1997.
9. Denis Caromel, Ludovic Henrio, and Bernard Serpette. Context inference for static analysis of java card object sharing. In *Proceedings e-Smart 2001*, volume 2140 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.
10. Zhiqun Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide*. The Java Series. Addison-Wesley, 2000.
11. Gennady Chugunov, Lars Åke Fredlund, and Dilian Gurov. Model checking multi-applet Java Card applications. In *Smart Card Research and Advanced Applications Conference (CARDIS'02)*, 2002.
12. Thomas Colcombet and Pascal Fradet. Enforcing trace properties by program transformation. In *27th symposium Principles of Programming Languages*, pages 54–66. ACM Press, 2000.
13. Karl Crary and Joseph C. Vanderwaart. An expressive, scalable type theory for certified code. In *International Conference on Functional Programming 2002*. ACM Press, 2002.
14. Damien Deville and Gilles Grimaud. Building an "impossible" verifier on a Java Card. In *USENIX Workshop on Industrial Experiences with Systems Software (WIESS'02)*, 2002.
15. U. Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *Symposium on Security and Privacy*. IEEE Computer Society Press, 2000.
16. Morrie Gasser. *Building a secure computer system*. Van Nostrand Reinhold Co., 1988.
17. Li Gong. *Inside Java 2 platform security: architecture, API design, and implementation*. The Java Series. Addison-Wesley, 1999.
18. James A. Gosling. Java intermediate bytecodes. In *Proc. ACM SIGPLAN Workshop on Intermediate Representations*, pages 111–118. ACM, 1995.
19. Pieter H. Hartel and Luc A. V. Moreau. Formalizing the safety of Java, the Java virtual machine and Java Card. *ACM Computing Surveys*, 33(4):517–558, 2001.
20. Nevin Heintze and Jon G. Riecke. The SLam calculus: programming with secrecy and integrity. In *25th symposium Principles of Programming Languages*, pages 365–377. ACM Press, 1998.
21. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP'97)*, number 1241 in Lecture Notes in Computer Science. Springer-Verlag, 1997.
22. Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proceedings Crypto '96*, number 1109 in Lecture Notes in Computer Science, pages 104–113. Springer-Verlag, 1996.
23. Markus Kuhn. Tamper resistance - a cautionary note. In *USENIX Workshop on Electronic Commerce proceedings*, pages 1–11, 1996.
24. Markus Kuhn. Design principles for tamper-resistant smartcard processors. In *USENIX Workshop on Smartcard Technology proceedings*, 1999.

25. Xavier Leroy. Java bytecode verification: an overview. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification, CAV 2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 265–285. Springer-Verlag, 2001.
26. Xavier Leroy. Bytecode verification for Java smart card. *Software Practice & Experience*, 32:319–340, 2002.
27. Xavier Leroy and François Rouaix. Security properties of typed applets. In J. Vitek and C. Jensen, editors, *Secure Internet Programming – Security issues for Mobile and Distributed Objects*, volume 1603 of *Lecture Notes in Computer Science*, pages 147–182. Springer-Verlag, 1999.
28. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, 1999. Second edition.
29. Sergio Loureiro, Laurent Bussard, and Yves Roudier. Extending tamper-proof hardware security to untrusted execution environments. In *USENIX Smart Card Research and Advanced Application Conference (CARDIS'02)*, 2002.
30. Greg Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. *Journal of Functional Programming*, 12(1):43–88, 2002.
31. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, 1999.
32. George C. Necula. Proof-carrying code. In *24th symposium Principles of Programming Languages*, pages 106–119. ACM Press, 1997.
33. George C. Necula and Peter Lee. Safe, untrusted agents using proof-carrying code. In Giovanni Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 61–91. Springer-Verlag, 1997.
34. François Pottier and Sylvain Conchon. Information flow inference for free. In *International Conference on Functional Programming 2000*, pages 46–57. ACM Press, 2000.
35. François Pottier and Vincent Simonet. Information flow inference for ML. In *29th symposium Principles of Programming Languages*, pages 319–330. ACM Press, 2002.
36. François Pottier, Christian Skalka, and Scott Smith. A systematic approach to static access control. In David Sands, editor, *Proceedings of the 10th European Symposium on Programming (ESOP'01)*, volume 2028 of *Lecture Notes in Computer Science*, pages 30–45. Springer-Verlag, 2001.
37. Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 2(4), 2000.
38. Zhong Shao, Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou. A type system for certified binaries. In *29th symposium Principles of Programming Languages*, pages 217–232. ACM Press, 2002.
39. Sean W. Smith and Steve Weingart. Building a high-performance, programmable secure coprocessor. Technical Report RC 21102, IBM Research, 1998.
40. Peter Thiemann. Enforcing security properties by type specialization. In *European Symposium on Programming 2001*, volume 2028 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.
41. Dennis Volpano and Geoffrey Smith. A type-based approach to program security. In *Proceedings of TAPSOFT'97, Colloquium on Formal Approaches in Software Engineering*, volume 1214 of *Lecture Notes in Computer Science*, pages 607–621. Springer-Verlag, 1997.
42. Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):1–21, 1996.

43. Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, 1993.
44. David Walker. A type system for expressive security policies. In *27th symposium Principles of Programming Languages*, pages 254–267. ACM Press, 2000.
45. Dan S. Wallach, Edward W. Felten, and Andrew W. Appel. The security architecture formerly known as stack inspection: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, 9(4), 2000.
46. Hongwei Xi and Robert Harper. A dependently typed assembly language. In *International Conference on Functional Programming '01*, pages 169–180. ACM Press, 2001.
47. Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Programming Language Design and Implementation 1998*, pages 249–257. ACM Press, 1998.