# Reflection as a Mechanism for Software Integrity Verification

DIOMIDIS SPINELLIS
University of the Aegean

The integrity verification of a device's controlling software is an important aspect of many emerging information appliances. We propose the use of reflection, whereby the software is able to examine its own operation, in conjunction with cryptographic hashes as a basis for developing a suitable software verification protocol. For more demanding applications meta-reflective techniques can be used to thwart attacks based on device emulation strategies. We demonstrate how our approach can be used to increase the security of mobile phones, devices for the delivery of digital content, and smartcards.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection

General Terms: Authentication

Additional Key Words and Phrases: cryptographic hash function, embedded device, message digest

## 1. INTRODUCTION

Information appliances and other devices with embedded software are becoming ever more sophisticated and widely used [Halla 1998]; increasingly in situations where their integrity is a prime concern. The complexity of their software and the rapidly evolving environment often mandate a field programming feature. Using this feature the owner or manufacturer can rapidly fix problems found after the product has left the factory, or even introduce new features.

Devices fitting the above description include mobile phones [Cummings and Heath 1999], pay-TV interfaces, sophisticated set-top boxes such as Web-TVs, credit card terminals, automatic teller machines, smart cards, routers, firewalls network computers, satellites, and space probes. As these devices often operate in a domain not fully controlled by the software's stakeholders their software can be compromised. An adversary might want

Author's address: Department of Information and Communication Systems, University of the Aegean, Karlovasi, GR-83200, Greece; email: dspin@aegean.gr.

to modify a device's software in order to avoid billing, bypass the device's anti-theft measures or intellectual property protection mechanisms, or implant a surveillance or denial of service device (Trojan horse). A mechanism is therefore needed to remotely verify the integrity of a device's controlling software.

In the following sections we will describe the problem, propose a solution based on software reflection, and outline some applications of our approach. The remainder of this paper is structured as follows: Section 2 outlines the problem we are addressing and the underlying assumptions; Section 3 describes the reflection-based solution strategy; Section 4 contains an analysis of our approach's vulnerabilities, while Section 5 provides examples of possible applications. In Section 6 we outline work related to our approach, and Section 7 concludes the paper with an assessment of the approach's contribution.

## 2. THE PROBLEM

The problem concerns the integrity verification of a device's controlling software. For risk analysis purposes we assume that the assets that are to be protected and the respective threats are those associated with consumer goods and services.

### 2.1 An Exemplar Threat

The global system for mobile communications (GSM), apart from the mobile subscriber identification and privacy security, provides functionality to secure the mobile terminal equipment (e.g. mobile phones) against theft. Each GSM terminal is identified by a unique International Mobile Equipment Identity (IMEI) number [Mouly and Pautet 1992]. A list of IMEIs in the network is stored in the Equipment Identity Register (EIR). The status returned in response to an IMEI query to the EIR is one of the following:

*White-listed*: The terminal is allowed to connect to the network.

*Grey-listed*: The terminal is under observation from the network for possible problems.

*Black-listed*: The terminal has either been reported stolen, or is not type approved (as the correct type of terminal for a GSM network). The terminal is not allowed to connect to the network.

A widely used mobile phone stores the IMEI number in an EEPROM (read-only memory that can be electrically reprogrammed). The phone also provides a "debug mode" which is accessed by setting an external connector pin to a high voltage and sending a specific command sequence through the phone's serial interface. The debug mode allows the programming of all the EEPROM contents, including the control software and the location used to store the IMEI number. Reportedly, as a security measure, the phone's software detects alterations to the IMEI number and resets all transceiver calibration data rendering the phone unusable. However, since all the phone's program

and data memory is field programmable using the phone's serial interface, a sophisticated thief could reprogram a stolen phone with software that did not perform the IMEI security checks and a new IMEI.

### 2.2 Definitions

Our problem can be formulated around the following entities:

*Client*: The client is a software-controlled field-programmable device whose software integrity the software stakeholder wishes to protect. Examples of such clients are the devices outlined in Section 1.

*Software Stakeholder*: The software stakeholder is an entity which has a business, legal, or regulatory interest in protecting the software's integrity. Examples of software stakeholders are the GSM operators, the owners of communication satellites, or the operators of pay-per-view systems.

*Server*: The server is a secure computer system under the control of the software stakeholder. The server can communicate with the client using a suitable communication protocol such as the remote procedure call.

*Adversary*: The adversary is an entity which has an interest in modifying the client's controlling software against the software stakeholder's will.

### 2.3 Assumptions

The approach we propose is meaningful when the entities we described operate within a framework of specific assumptions. These assumptions, although they cannot be guaranteed, are realistic under a risk-analysis view [Pfleeger 1996, p.15]: they hold true for a wide set of typical assets that are to be protected and associated threats. Examples of such typical asset-threat pairs include the illegitimate operation of a mobile phone by a well-connected street criminal, or the unauthorized viewing of a pay-per-view movie by a computer science student.

*Assumption 1.* The adversary can reverse engineer the client's software and hardware. We assume that the client's software, including software that controls the software upload session, is stored in unprotected memory devices that can be read, reverse engineered, and modified. This assumption is typically valid due to cost restrictions imposed by tamper-resistant hardware and the low cost associated with modifying many firmware storage technologies such as EEPROMs, flash ROMs, and CD-ROMs.

*Assumption 2.* The adversary can modify the client's software at will. As we will explain in Section 3.1.1 it is not feasible to protect the client's upgrade procedure when Assumption 1 holds true.

*Assumption 3.* The adversary cannot modify or substitute the client's hardware. Under the risk analysis regime we outlined, the cost of this operation would be prohibitive compared to the potential benefits.

*Assumption 4.* The adversary cannot mount a man in the middle attack. This mode of attack is often difficult and costly; its cost is comparable to that of the hardware substitution (Assumption 3). In addition, if such an attack is mounted, the integrity of the client's software will be the least of the software stakeholder's concerns.

*Assumption 5.* The effective entropy of the machine representation of the client's functioning software is for practical purposes equal to the device's available storage. This assumption means that the client is not allowed to contain empty or low-entropy memory contents. To realize this assumption, empty memory can be filled with random data. In addition, low entropy memory contents can be compressed to increase their entropy. For cases where this approach is not feasible, we outline an extension to our approach in Section 3.3, which obviates the need for this assumption.

## 3. SOLUTION STRATEGY

A number of solutions to the problem we outlined turn out to offer relatively weak protection or be infeasible in practice. In the following paragraphs we briefly outline these approaches because each one of them contains a part of the solution we propose.

### 3.1 Non-Solutions

3.1.1 *Programming Authentication.* Protecting the client's software integrity from unauthorized modification can be viewed as a standard access control problem solved through a suitable identification and authentication (I&A) service. I&A is the twofold process of first identifying an entity and then validating the identity of this entity. In order to implement an authentication mechanism, one must determine what information will be used to validate the potential user. Whatever that information is, it can be described by one or more of the following categories:

—secret information (something the user knows),

—possession of a device (something the user has),

—biometrics (something the user is), or

—location-based authentication (somewhere the user is).

Thus, the client could implement an I&A protocol for the software upgrade disallowing unauthorized software modifications. Unfortunately, since the adversary can reverse-engineer the client's software, he can also reverse engineer the client's authentication protocol and keys, and consequently obtain access to the software modification functionality.

3.1.2 *Code Checksum.* One other approach could involve the calculation of a checksum of all the client's program memory contents. The server could then periodically query the client for that checksum and verify it against a locally stored copy. The obvious weakness of this approach is that rogue

software uploaded to the client by the adversary could implement a dummy version of the checksum function that would send back the checksum of the original software.

3.1.3 *Code Transfer*. Although the adversary could implement a dummy checksum calculation function, it would be impossible for him to store a complete copy of the original software in conjunction with the modified version due to the constraint imposed by Assumption 5. The server could therefore verify the client's software integrity by requesting the client to transmit a copy of its operating software. The problem of this approach is the bandwidth required to implement it. Many clients communicate over a low bandwidth channel, yet contain a large amount of controlling software. Transmitting a copy of the software to verify its integrity could consume large amounts of time over a potentially costly communications medium.

Our approach is based on a synthesis of the last two solution attempts described above.

### 3.2 Reflection-Based Verification

Our solution is based on having the client's software respond to queries about itself. The theoretical basis for this course is *reflection*. The foundation for the concept of reflection is Smith's reflection hypothesis [Smith 1982]:

> "In as much as a computational process can be constructed to reason about an external world in virtue of comprising an ingredient process (interpreter) formally manipulating representations of that world, so too a computational process could be made to reason about itself in virtue of comprising an ingredient process (interpreter) formally manipulating representations of its own operations and structures."

Based on this hypothesis, reflective programs can access, reason about, or alter their interpretation.

An important property of reflection is the casual connection between the system's internal structures and the domain they represent [Maes 1987]. It is this property of reflection—that requires the connection to internal structures and the represented domain to be behavioral rather than just functional—that we exploit to provide an authentication mechanism. Specifically, we reason that the internal representations of a system can be used to authenticate its external, functional properties.

Although the semantics of reflection in programming languages are often complicated, our requirements are modest and can in most cases be satisfied as a normal part of the system's implementation, i.e., without imposing special needs on the implementation language or environment. For our application it is sufficient to amend the client's program with the ability to access its own internal, machine-code representation. Thus, the theory-relative self-knowledge implied by reflection boils down in our case to read-only access of the program's internal representation. If the client's program and its associated data are stored in memory locations $[0 - L]$,

the program needs to be able to bind the name of a memory location $M$ with the value of its contents $V$ and obtain that value. The viability of this operation is a consequence of a shared code-data (von Neumann) architecture and made possible in languages that support the use of unrestricted pointers such as C and C++. At a higher level, access to an application's code memory can be supported through operating system abstractions such as system calls or special block device files [Yokote 1992]. In order to be able to verify the integrity of the device's software, stakeholder Bob ($B$) amends the protocol the serve uses to communicate with the client device ($D$) by adding the the following message:

$$B \rightarrow D : H(S, E) \tag{1}$$

This message requests from the device to compute and send back a cryptographic hash [Pieprzyk and Sadeghiyan 1993] (message digest) such as RIPEMD-160 [Dobbertin et al. 1996] of its program storage locations ranging from $S$ to $E$. When the device receives this message it responds with the tuple:

$$D \rightarrow B : (V, H_{(V, S, E)}) \tag{2}$$

which contains its operating program version $V$ and the computed hash value $H$ for that given version. In order to verify the integrity of the device's software, $B$ needs to choose two random integers $M_1$ and $M_2$ that satisfy the following condition:

$$0 \leq M_2 \leq M_1 \leq L \tag{3}$$

The following message exchange will then take place:

$$B \rightarrow D : H(0, M_1) \tag{4}$$

$$D \rightarrow B : (V, H_{(V, 0, M_1)}) \tag{5}$$

$$B \rightarrow D : H(M_2, L) \tag{6}$$

$$D \rightarrow B : (V, H_{(V, M_2, L)}) \tag{7}$$

$B$ will then retrieve from the server's secure database a copy of the same software version $V_L$, calculate the corresponding hash values, and compare them against the values sent by $D$:

$$H_{(V_L, 0, M_1)} \stackrel{?}{=} H_{(V, 0, M_1)} \tag{8}$$

$$H_{(V_L, M_2, L)} \stackrel{?}{=} H_{(V, M_2, L)} \tag{9}$$

If the two values do not match, $B$ knows that the software has been tampered with and can refuse service to that device, or reprogram it with the correct software.

The verification procedure we outlined overcomes the problems described in Section 3.1. Specifically, unauthorized software modifications of any part of the software, including the part that implements the verification protocol, will be detected, as the modified software will be covered by the input range of at least one of the two hash functions and the respective function will yield a result different from the one calculated by the server on the original software. In addition, as the two positions used to delimit the ranges of the hash functions are specified dynamically as random integers during the protocol operation, the rogue software cannot precalculate and store all possible responses to the hash value query message 1; storage availability for storing the precalculated values is restricted by Assumption 5. Finally, the amount of information transferred using the outlined protocol (messages 4–7) is modest, totaling less than 64 bytes in both directions for a 160 bit hash function; it is therefore practical to implement it even over low bandwidth channels.

The hash function used for evaluating the return value should be a cryptographicaly secure function such as RIPEMD-160. For a given program version $V$ and a program of length $L + 1$ this function maps the $2(L + 1)$ different $(S, E)$ tuples that can be requested by $B$ onto the corresponding hash values. The properties of this function should preclude its emulation by any implementation other than one that has access to the original program code.

As $M_2 \leq M_1$ the hashes computed will span over the entire program storage locations of $D$. With RIPEMD-160, hash function implementations computing hashes at 19.3 Mbit/s (portable C implementation) up to 39.9 Mbit/s (hand-tuned x86 assembly) on 90 MHz Pentium machines [Bosselaers et al. 1996]. The performance of our approach is within the practical limits of both high-end (e.g. 2Mbyte ROM 100MHz processor information appliances) and low-end (e.g. 20Kbyte ROM 10MHz processor smartcards) hardware.

### 3.3 Meta-Reflective Extension

Our Assumption 5 specifies that the effective entropy of the machine representation of the client's functioning software is for practical purposes equal to the device's available storage. This assumption is not always easy to satisfy in practice. Physical memory contents are typically of low entropy and thus compressible [Douglas 1993]. An adversary could therefore compress the original software into an unused memory area and then execute and digest the compressed version using on-the-fly decompression techniques.

A way around this attack is based on the difficulty of predicting and monitoring a modern machine's processor behavior. Although in the absence of external interrupts (which can be disabled) a processor operates in

a deterministic fashion, the exact modeling and prediction of a modern processor's state after a calculation is nowadays impossible without a low-level simulator and intimate knowledge of the processor's architecture. The difficulty arises in sophisticated pipelined processor architectures [Hennessy and Patterson 1996] from the interdependencies of the multiple functional units, the many levels of cache, and branch predictors all dynamically changing their behavior as the program executes [Spinellis 1999].

Processor state can be set to a known value before the message digest calculation and returned as part of the result to the server. The server can then query a client known to contain valid software for the same values and compare the results. Examples of processor state that can be queried are the contents of the processor's cache or the processor's clock-cycle granular "performance counter." On-the-fly decompression will be immediately revealed by examining the number of clock-cycles that were required to calculate the hash using the performance counter. In addition, the behavior of the cache is affected by its $n$-way associativity—even a shift of the client's software to a different address will probably affect the cache's behavior during the program's operation.

The meta-reflective extension to the protocol we propose can be implemented by amending the message exchange 1 (the reply of $D$) to contain a representation of the state $T_D$ at the end of the calculation of $H_{(V, S, E)}$:

$$D \rightarrow B : (V, H_{(V, S, E)}, T_D) \tag{10}$$

$B$ will then need to perform the same calculation on a second device $D'$ known to contain an authentic version $V$ of the software and compare the corresponding results:

$$H_{(V_L, S, E)} \stackrel{?}{=} H_{(V, S, E)} \tag{11}$$

$$T_{D'} \stackrel{?}{=} T_D \tag{12}$$

Although the precise details for obtaining $T$ are hardware architecture and implementation dependent, the general outline of this procedure is as follows:

(1) Disable external interrupts.

(2) Set the system state to a known initial value (e.g. clear the cache and the clock counter).

(3) Perform the hash calculation.

(4) Record the final system state $T$.

(5) Re-enable external interrupts.

## 4. VULNERABILITY ANALYSIS

An obvious vulnerability of the described scheme stems from the possibility of storing the original program code into locations not read by the hash function (e.g. unused memory) and using that code for generating the hash reply. A practical implementation of this approach could only store the original parts of the modified portions in unused memory and adjust the digest function accordingly. This vulnerability can be overcome by filling all unused program memory space with random data elements and extending the program space that can be hashed to include those elements. A refinement of the above attack involves either storing the original program code into unused data storage locations, or compressing the original and the modified program code to fit into the program memory together. In the latter case the modified program code is executed using either interpretive or on-the-fly decompression techniques. Both attacks can be avoided by using meta-reflective techniques or by structuring the software and the communications protocol in a way that will make such attacks detectable. As an example, the end that communicates with the device can measure the reply latency for a given version to detect the slowdown imposed by decompression or interpretation.

Our approach is particularly vulnerable to a man in the middle attack. Specifically, an adversary can store a valid program in another pristine device $D_p$; the compromised device $D_c$ can relay the server authentication requests to $D_p$ and send back the response of $D_p$ to the server. Although we assumed that such an attack will typically be economically unattractive a simple solution exists: in many cases where communication takes place over a network with stable and controlled timing properties—such as the applications described in the following section—the attack can be detected by timing differences in the reply of $D_c$.

## 5. APPLICATIONS

In the following paragraphs we outline some applications of reflective software integrity verification in the areas of mobile phones, intellectual property protection, and smartcard-based systems.

### 5.1 Mobile Phone Software Validation

The attack against the firmware of a mobile phone described in Section 2.1 can be made considerably more difficult by making the phone's software sufficiently reflective. As an example, copies of the software of registered mobile phones could be kept in a secure database. Every time a mobile phone identifies itself to a base station, the base station could send it a software verification command and compare the phone's response against the result calculated using the software copy in the server's database. Phones failing the software verification would not receive service, making them unusable.

### 5.2 Digital Content Intellectual Property Protection

One other deployed technology that fits the usage scenarios of our approach is the one used for pay-per-view billing of digital content. The scheme used by the Digital Video Express DVIX [Mehrotra 1999] discs allows consumers to view a film they purchase on a cheap DVD-like disc within 48 hours after first hitting the play button. Additional viewing sessions or "for life" viewing can be selected from a menu and then paid by credit card. The billing information is transferred overnight from the device to the company operations center using a modem built into the device. It is natural to assume that the device's software will receive considerable interest from adversaries wishing to be able to view the disks without getting billed. Assuming that in order to continue to be able to view disks some keys will need to be supplied by the system's server (thus prohibiting the stand-alone operation of a cracked system), the device's communication protocol can be enhanced to respond to reflective commands in order to guard against unauthorized modifications to the device's software. Similar strategies can be used to protect pay-per-view TV set-top boxes as well as the next generation of game consoles that will contain a built-in modem. Attacks against set-top boxes and game console software are already a reality; attacks against the DirectTV system were widely publicized; we are also aware of a modification performed on Sony Playstation consoles to make them work with illegal game copies on CD-R.

### 5.3 Smartcard Verification

Many smartcard applications such as electronic purses, credit cards, and phone cards depend on the smartcard's integrity to protect the stakeholder's financial interests against fraud. Advanced smartcard models integrate a microprocessor, RAM, ROM, an operating system, and application programs. Although smartcards are relatively tamper-proof devices, the possibility of attacks based on reverse engineering, fault injection [Maher 1997], or misappropriation of confidential design information cannot be excluded. If a smartcard and its memory contents are successfully reverse engineered an adversary could implement a contraband version of the smartcard, or a corresponding emulator. A practical defense against contraband smartcards based on the same hardware as legitimate ones can be the extension of the card's protocol with reflective verification techniques. In addition, meta-reflective techniques can be used to guard against attacks based on smartcard emulators.

## 6. RELATED WORK

Reflective capabilities were originally proposed to provide computational processes with the ability to reason both about a given domain and about their reasoning process over that domain [Smith 1982, p. 3]. The use of the reflection paradigm quickly spread beyond the area of programming language design and was applied in diverse areas such as windowing systems [Rao 1991], operating systems [Yokote 1992], and distributed systems

[Edmond et al. 1995]. The power of reflection allows the implementation of open, flexible, and adaptable systems; a requirement in the areas it has been applied, and—in our view—an asset for security applications operating outside the absolute control of their stakeholders.

Reflective techniques for verifying the integrity of software are often used as part of a device's power-up self-test procedure. This approach typically guards against hardware malfunctions of the memory system and therefore uses simple checksums instead of cryptographically secure hash functions. As an example, the BIOS ROM of the original IBM PC contains code that calculates and verifies the ROM checksum at power-up time [IBM Corporation 1983, p. A-6]. Similar self-checking techniques have also been proposed for protecting programs against viruses [Cohen 1990]. Both types of protection cannot withstand attacks that can reverse engineer and modify the program that performs the check and, at the same time, is being checked.

A protocol similar to the one we outline has been proposed to supply a proof of existence of a particular block of data [Williams 1994]. When $B$ wants to prove to $A$ that it possesses a particular block of data $D$ (the exemplar case mentioned entails auditing of off-site backup services), $A$ sends to $B$ a random number $R$. When responding, $B$ calculates and returns to $A$ the digest of $RD$. $A$ can then compare this value against a pre-calculated and stored value.

Finally, the use of hashes together with user-supplied passwords has been proposed as part of a method to securely boot workstations operating in a hostile environment [Lomas and Christianson 1995]. Under this approach *collision-rich hashing* is used in conjunction with a low-entropy user password to protect the kernel against off-line password guessing attacks.

## 7. CONCLUSIONS

Reflection—the software reasoning about its operation—can provide a software verification basis for a large, commercially important, class of products and services. In the previous sections we formalized its use and described a verification protocol that can be used to verify that the software embedded in a device has not been modified. The reflective techniques can be extended at a meta level by reasoning about the software's reasoning process thus providing an extra layer of security. The applications of our approach include software verification in personal communication devices, intellectual property protection, and smartcards. We believe that the increasing convergence of communication and information devices in a domain where security, privacy, and financial interests are often controlled by software will provide a fertile ground for applying many reflective techniques similar in spirit to the ones we described.

## REFERENCES

BOSSELAERS, A., GOVAERTS, R., AND VANDEWALLE, J. 1996. Fast hashing on the Pentium. In *Proceedings of the 16th Annual International Conference on Advances in Cryptology* (CRYPTO '96, Santa Barbara, CA, Aug.), LNCS 1109, N. Koblitz, Ed. Springer-Verlag, New York, NY, 298–312.

COHEN, F. 1990. Implications of computer viruses and current methods of defense. In *Computers Under Attack: Intruders, Worms, and Viruses*, P. J. Denning, Ed. ACM Press, New York, NY, 381–406.

CUMMINGS, M. AND HEATH, S. 1999. Mode switching and software download for software defined radio: The SDR forum approach. *IEEE Commun. Mag. 37*, 8 (Aug.), 104–106.

DOBBERTIN, H., BOSSELAERS, A., AND PRENEEL, B. 1996. RIPEMD-160: A strengthened version of RIPEMD. In *Proceedings of the Third International Workshop on Fast Software Encryption* (Cambridge, UK), LNCS 1039, D. Gollman, Ed. Springer-Verlag, New York, NY, 71–82.

DOUGLAS, F. 1993. The compression cache: Using on-line compression to extend physical memory. In *Proceedings of the Third USENIX Conference* (Anaheim, CA, Jan.), USENIX Assoc., Berkeley, CA, 519–529.

EDMOND, D., PAPAZOGLOU, M., AND TARI, Z. 1995. An overview of reflection and its use in cooperation. *Int. J. Intell. Coop. Inf. Syst. 4*, 1, 3–44.

HALLA, B. 1998. How the PC will disappear. *IEEE Computer 31*, 12 (Dec.), 134–136.

HENNESSY, J. L. AND PATTERSON, D. A. 1996. *Computer Architecture: A Quantitative Approach*. 2nd ed. Morgan Kaufmann Publishers Inc., San Francisco, CA.

IBM CORPORATION. 1983. IBM Personal Computer Technical Reference Manual.

LOMAS, M. AND CHRISTIANSON, B. 1995. To whom am I speaking: Remote booting in a hostile world. *IEEE Computer 28*, 1 (Jan.), 50–54.

MAES, P. 1987. Concepts and experiments in computational reflection. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA '87, Orlando, FL, Oct. 4–8), N. Meyrowitz, Ed. ACM Press, New York, NY, 147–155.

MAHER, D. P. 1997. Fault induction attacks, tamper resistance, and hostile reverse engineering in perspective. In *Proceedings of the First International Conference on Financial Cryptography:* (FC '97, Anguilla, British West Indies, Feb.), LNCS 1318, R. Hirschfeld, Ed. Springer-Verlag, New York, NY, 109–121.

MEHROTRA, R. 1999. In the news: Dueling over DVD and DIVX. *IEEE MultiMedia 6*, 1 (Jan.-Mar.), 14–19.

MOULY, M. AND PAUTET, M.-B. 1992. The GSM System for Mobile Communications (Published by the authors).

PFLEEGER, C. P. 1997. *Security in Computing*. Prentice-Hall, Inc., Upper Saddle River, NJ.

PIEPRZYK, J. AND SADEGHIYAN, B. 1993. *Design of Hashing Algorithms*. LNCS 756, Springer-Verlag, New York, NY.

RAO, R. 1991. Implementational reflection in Silica. In *Proceedings of the European Conference on Object-Oriented Programming* (ECOOP '91, Geneva, Switzerland, July 15–19), LNCS 512, P. America, Ed. Springer-Verlag, New York, NY, 251–267.

SMITH, B. C. 1982. Procedural reflection in programming languages. Ph.D. Dissertation. MIT Laboratory for Computer Science, Cambridge, MA.

SPINELLIS, D. 1999. Software reliability: Modern challenges. In *Proceedings of the Tenth European Conference on Safety and Reliability* (ESREL '99, Munich-Garching, Germany, Sept.), G. I. Schuëller and P. Kafka, Eds. A. A. Balkema, Rotterdam, The Netherlands.

WILLIAMS, R. N. 1994. An introduction to digest algorithms. ftp://ftp.rocksoft.com/clients/rocksoft/papers/digest10.ps.

YOKOTE, Y. 1992. The Apertos reflective operating system: The concept and its implementation. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA '92, Vancouver, British Columbia, Canada, Oct. 18–22), J. Pugh, Ed. ACM Press, New York, NY.

Received: May 1999; revised: October 1999; accepted: December 1999