Software Tampering Detection using AOP and mobile code

Paolo Falcarin [1], Mario Baldi [1], Daniele Mazzocchi [2]

[1] Politecnico di Torino, Dipartimento di Automatica e Informatica, Corso Duca degli Abruzzi, 24, Torino, Italy
[2] Istituto Superiore Mario Boella, Via Boggio 61, Torino, Italy
Paolo.Falcarin@polito.it, Mario.Baldi@polito.it, mazzocchi@ismb.it

Abstract

Assuring that a given code is faithfully executed with defined parameters and constraints on an un-trusted host is an open problem, which is especially important in the context of computing over communications networks. This work evaluates applicability of Aspect-Oriented Programming (AOP) to the problem of remotely authenticating code during execution, which aims at assuring that the software is not maliciously tampered prior to and during execution. A flow of idiosyncratic signatures is continuously generated and associated to data transmitted by a function that is encapsulated in an aspect and whose execution is subordinated to the proper execution of the software being authenticated. The flow of signatures is validated by a remote component.

1. Introduction

Among the very broad range of security issues, this paper investigates how to apply Aspect Oriented Software Development (AOSD) techniques to implement software-tampering detection in applications running on an un-trusted host. There are many situations in which it is desirable to protect a piece of software from malicious tampering once it gets distributed to a user community (examples include time-limited evaluation copies of software, password-protected access to unencrypted software, controlled playback of copyright protected material, e-voting and e-commerce systems) or even when running on a server (e.g., systems handling critical information and financial transactions).
In general, software, especially in the context of data networks, suffers from some inherent problems. These include modifications by an either malicious or inadvertent user, malware distribution (e.g., viruses and "Trojan horses"), and the use of malicious software remotely for penetration, intrusion, denial-of-service (DoS), and distributed DoS (DDoS). For example, a rogue user may manipulate the code of a given protocol (such as TCP) and gain an unfair advantage in using network bandwidth.
Tamper resistance is the set of methodologies for protecting software or hardware from unauthorized modification, distribution, and misuse [9]. One important technique is integrity checking and in particular self-checking, in which a program, while running, checks itself to verify that it has not been modified. We distinguish between *static* self-checking, in which the program checks its integrity only once, during start-up, and *dynamic* self-checking, in which the program repeatedly verifies its integrity at run time. Self-checking alone is not sufficient to robustly protect software from tampering since the self-checking function itself can be removed or inhibited.
The level of protection from tampering can be improved by using techniques that slow down reverse engineering, such as customization and obfuscation, techniques that prevent using debuggers and emulators, and methods for marking or identifying code, such as watermarking. These techniques reinforce each other, but they do not make it bulletproof [8].
We think that whichever self-checking technique, bundled within the application, can be identified and disabled by an attacker with enough knowledge, time, and reverse engineering tools. We noticed that current self-checking techniques rely on static code checkers whose position is hidden in the application and whose behavior is obfuscated or complex to understand.
Hence, the presented solution extends the power of code checkers in two ways: it adds *remote verification* that self-checking has been performed and continuous replacement of (critical parts of) the self-checking code.
Software tampering detection is indeed a crosscutting concern, because of its pervasive nature with regard to the business logic in an application. Aspect Oriented Programming (AOP), being an emerging technology promoting advanced separation of concerns, can be used to ease the design of different self-checking techniques and, in our approach, it is used to modularize self-checking code in an aspect whose behavior can be continuously updated with mobile code.

This paper describes the design and implementation of such a solution and its dynamic self-checking mechanism that can raise the level of tamper-resistance protection against an adversary with static analysis tools and knowledge of our algorithm and most details of our implementation.
We begin in Section 2 with a brief discussion of related work. In Section 3 we address our design objectives we used to create techniques for remote authentication of code execution. Section 4 presents an overview of the proposed self-checking mechanism based on AOP, and possible threats are detailed in section 5. Finally, Section 6 concludes with a brief discussion of directions for future work.

2. Related Work

There has been a significant amount of work on the problem of executing un-trusted code on a trusted host computer [10, 11]. The field of tamper resistance is the dual problem: running trusted code on an un-trusted host. Although of considerable practical value, there has been little work done on this problem. Most of the work reported in the literature is ad hoc. It is not clear that any solution exist that has provable security guarantees and with measurable effectiveness. We present a list of some important work related to self-checking technology: for more details see [9, 2].
Obfuscation attempts to thwart reverse engineering by making it hard to understand the behavior of a program through static or dynamic analysis. Obfuscation techniques tend to be ad hoc, based on ideas about human behavior or methods aimed to defeat automated static or dynamic analysis. Collberg, et al. [2] presented classes of transformations to a binary that attempt to confuse static analysis of the control flow graph of a program. Wang, et al. [12] also proposed transformations to make it hard to determine the control flow graph of a program by obscuring the destination of branch targets and making the target of branches data-dependent.
Customization takes one copy of a program and creates many very different versions. Distributing many different versions of a program stops widespread damage from a security break since published patches to break one version of an executable might not apply to other customized versions; then each instantiation of a protected program may be different [13].
Software watermarking, which allows tracking of misused program copies, have been proposed in different ways: Collberg and Thomborson [14] provide a survey of research and commercial methods: they make the distinction between software watermarking methods that can be read from an image of a program and those that can be read from a running program.
Self-checking is an essential element in an effective tamper-resistance strategy. Self-checking detects changes in the program and invokes an appropriate response if change is detected. This prevents both misuse and repetitive experiments for reverse engineering or other malicious attacks. Aucsmith [13] presents a self-checking technology in which embedded code segments verify the integrity of a software program as the program is running. These embedded code segments check that a running program has not been altered, even by one bit.
On the other side, network security in all its aspects has become more crucial in recent years, in terms of protecting data travelling through the network and authenticating communicating entities (see for example [5, 6, 7]). However, no work has been done for assuring the integrity of the software implementing the communicating entities and generating the data. The presented work aims at addressing this issue, trying to use aspect-oriented programming and dynamic code downloading to enforce code-checkers relying on network security.

3. Remote verification: the TrustedFlow™ Protocol

The TrustedFlow protocol [15], as shown in Figure 1, is based on a Trusted Flow Generator (TFG) in the entrusted code of the program deployed on an un-trusted host, and a Trusted Tag Checker (TTC) function on a trusted host, i.e. another computer or as part of some network interface (e.g., firewall, gateway), acting as entrusting entity. The TFG is a module that generates a pseudo-random sequence of n-bit tags (idiosyncratic signature) depending on a random seed and possibly on the information to be transmitted.
The n-bit tags are interleaved in the sequence of messages (e.g., inside data packet headers) that are sent through the network. At the destination or at an intermediate node within the network the validity of the pseudo-random sequence of n-bit tags is checked and verified by the TTC function. A valid pseudo-random sequence of n-bit tags verifies that the sending computer has used the appropriate software (programs and parameters). Consequently, the TTC function accepts and/or forwards only data packets from well-behaved sources.

It is important that the algorithm used to generate the idiosyncratic signature does not require a strong synchronization between TTC and the TFG. One possible idea currently under investigation is the use of a block cipher in counter mode with the inclusion the value of the counter among the data transmitted between the TTC and the TFG.
Moreover, a MAC (Message Authentication Code) of the data packet including the current idiosyncratic signature can be generated to protect against the tampering with the data associated to a valid signature.

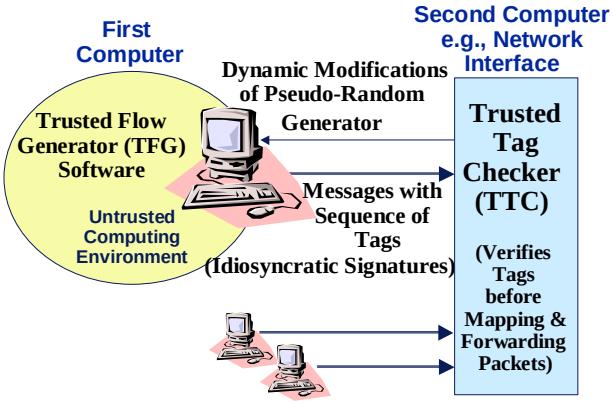

Figure 1: TrustedFlow architecture

TrustedFlow is based on the following two principles.

- **Interlocking** – original executable software modules (original function) are combined together with the idiosyncratic (pseudo-random) signature generator (which is used to generate an unpredictable flow of tags) in such a way that proper execution of interlocked code in necessary to proper functioning of the generator.
- **Program Hiding** – the combined module is prepared in such a way as to ensure that reverse engineering is difficult enough so that it becomes functionally infeasible. Periodic replacement of (parts of) the combined module can be used to cope with the limited robustness of program hiding.

This work shows how AOP can be used to realize self-checking — on which interlocking is based – and periodic replacement — on which hiding is based. Approaches do exist that hide some sort of self-checker in the application running on an un-trusted host to make it tamper-proof. Such self-checking module is subject to reverse engineering attacks. Our solution is not vulnerable to such attacks thanks to the following two additional features

1. Both the self-checking algorithm and the data used by this algorithm to verify software integrity are variants. Once these features are variants and modularized with AOP they can become dynamically variable in time. Therefore, the TTC can periodically release new version of the TFG, making former versions invalid. Our approach is based on the assumption that even if attacker were able to understand and crack the application, the time needed to pursue this goal would be anyway longer than or even just comparable to the validity of the attacked TFG code.
2. The TTC, which is running in a trusted environment, makes sure that the self-checking module is not bypassed or isolated by invalidating data that has not been associated to valid tags as generated by the self-checking module.

The goal of the proposed methodology is to enable detection of the following attacks:

- Modification of values of fields and constants outside a specified range.
- Execution of tampered versions of software for malicious goals.
- Substitution of relevant operations in the application to modify application behavior.
- Replacement of code with a cracked version for malicious goals.
- Bypass of code parts verifying licenses or billing.

The proposed architecture is depicted in Figure 2.

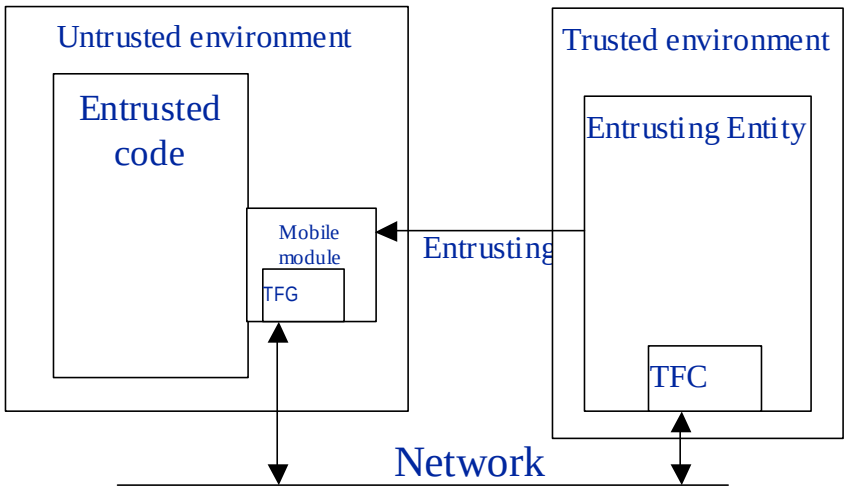

Figure 2. The TrustedFlow approach

The TFG is executed within a *mobile module* that is not bundled within the application but it is downloaded at runtime to make reverse-engineering attacks more difficult. Obfuscation techniques thwart fast reverse-engineering attacks. . Even in case of de-compilation of the mobile module using ad-hoc packet sniffers and/or memory dump, the periodic substitution of the mobile code and its own checking algorithms, limits the validity of manual reverse-engineering, as more as the change period become smaller.
The realization of the proposed system can be divided in four sub-problems:

1. *network level*: defining message exchanges and a header format suitable to carry tags and supplementary information (e.g., sequence number and tag flow identifier) together with data generated by the entrusted module necessary for the TTC to be able to verify tags and periodically replace the mobile module..
2. *security level*: defining cryptographic aspects of tag calculation, starting from an initial seed hardwired in the mobile module.
3. *tag interlocking with code execution*: the choice of input data for tag calculation must be driven by two goals: verifying that sensible data are not tampered, and verifying that execution sequence is compliant with the original specification used by TTC to perform validation
4. *mobility*: identifying techniques for dynamically replacing and installing the mobile module ensuring proper interaction with the entrusted code.

4. Tampering detection with AOP and mobile code

The presented work focuses on the mobility sub-problem and this section analyses possible applications of AOP to the implementation of the TFG. We analyzed the mobile code issue in the Java environment and identified three different approaches relying on these candidate technologies: software agents, static AOP and dynamic class-loading, dynamic AOP platforms.
With static AOP we means tools like AspectJ [17], which weaves aspects at compile-time, while dynamic AOP platforms are able to weave and unweave aspects at run-time.
Agent platforms allow mobile code to be executed on a remote un-trusted host running an application: the agent runs in a different process and interacts with the application through a well-defined interface dependent on the chosen agent platform. After a preliminary analysis we believe that, unlike AOP techniques, agents cannot have full control of application code.
Static AOP tools can be coupled with dynamic class loading to obtain dynamic insertion of code extensions in the application running in an un-trusted host.
In this case AOP, an application can be woven with a set of aspects that intercept all relevant method invocations and field accesses: the advice code of each aspect can get actual parameters and field values using AOP features and these data can be subsequently used to check some validity constraints and eventually disable tag generation if an unexpected value is obtained.
AOP can thus insert “hooks” in relevant parts of application code that call the advice method of different aspects; at runtime the set of relevant data (e.g. fields values, actual parameters of an invoked method, application class definitions determined with reflection) are available to the advice code. Among this set of data, the mobile code can select a subset to perform validity checking and these data can be used as input for the tag generation algorithm, in conjunction with

a key within the mobile module, i.e. the dynamically inserted code that is periodically renewed by the entrusting entity.
Moreover if different aspects and different types of mobile code are used they can validate the application and they may validate each other.
Using dynamic AOP platforms [16] maintains all the advantages of the static AOP approach, but there is no more presence of "hooks" in the application code (possible starting point for a "replacement" attack), because these hooks are determined at runtime by the platform, depending on pointcut definitions included in the dynamically downloaded aspect. Also, dynamic AOP enables to download and withdraw a set of different aspects, each one making a specific check on application code, which thwarts attackers.
Using a dynamic AOP platform or dynamic code downloading means that the prototype relies on an external, un-trusted support for mobile code installment and execution. Reliance on such un-trusted support could make the whole system vulnerable, for a limited time.
However, we do not consider this being a problem for our purposes since our aim is to demonstrate that our approach can be immune to this problem. In fact if the attacker is able to modify the AOP platform, in order to disable mobile code download (and the TFG module here contained), then tags cannot be generated and TFC (in the Entrusting Entity) will discover tampering attempt and react. In future, when dynamic AOP platform will become more secure, then our approach could be improved.
As the Java platform does not allow application access to the code segment, code verification techniques cannot be applied. If the application is implemented in C++, the aspect's advice code can calculate a hash of the code segment at run-time. The mechanism could detect the change of a single bit in any non-modifiable part of the program, as the program is running and soon after the change occurs. This helps to detect an attack in which the program is modified temporarily and then restored after unspecified behavior occurs. Our approach modularizes self-checking code in aspects that can be independently replaced or modified, making future experimentation and enhancements easier, and making extensions to other executable formats easier.
Other benefits of a Java based solution are hardware platform independence and easy integration with other tamper-resistance methods techniques like customization: different version of the application and related aspects can be easily generated acting on aspect pointcuts definitions. The power of pointcuts composition rules in AOP is suitable for a flexible management and distribution of self-checking code in a large code base.

5. Possible threats

The fundamental purpose of a dynamic self-checking mechanism is to detect any modification to a program as it is running, and upon detection to trigger an appropriate response. In this approach such response involves inhibiting the generation of the idiosyncratic signature. Using the TrustedFlow protocol any software tampering on the un-trusted host should be remotely detected by the TTC, whose response is blocking any further network communication coming from the suspected host. This section discusses possible threats to the proposed approach and its robustness to such threats.
A possible attack is to identify the mobile module using packet sniffers and/or memory dump: once identified and decompiled the mobile module the attacker can understand the TFG behavior and disable checking activities, but sending the correct information to the TTC. It is unlikely that such a complex attack, that is likely to require discovering the behavior of all the aspects, can be successfully completed before their validity expires.
The two general attacks on a software self-checking mechanism are *discovery* and *disablement*. Discovery methods and our approach for preventing or thwarting these methods, follow.
Among discovery attacks, *static inspection* made with automated inspection tools (by a program) can be defeated by our approach, by the dynamic change of the TFG encapsulated in the mobile module.
Off-the-shelf *dynamic analysis tools*, such as debuggers and profilers, pose a threat to our self-checking approach, in particular when a dynamic AOP platform is used. Moreover when static AOP with dynamic class loading is used, the aspect's advice code invocations are identifiable in the code.
Obfuscation of the TFG and the limited time validity of its algorithm can limit the effectiveness of the attack. The attack is fully voided if the time needed to succeed is longer than the current TFG time validity.
The self-checking mechanism consists of a number of aspects, each testing a small set of code structure and properties. An attacker, having discovered one such aspect, could look for others by

searching for similar code sequences. Customization of aspect code can help, so that generalizing from one to others is difficult: e.g. multiple aspects, each one performing a different check (computing a different part of the seed from a different subset of attribute values or code structure), could implement a basic customization technique.
Disablement attacks can be defeated by the remote validation made by the TFC of the signed packets coming from the current TFG in the application.
One possible disabling attack is to modify one or more aspects so that they fail to signal a modification. If no data are sent by the TFG, the TTC deduces that a tampering has been carried out, and network connectivity of the application is blocked.
A possible improvement may be based on an overlapping coverage, so that each aspect is validated by several others. Disabling one or more of the aspect advices by modifying them will produce detection of these changes by the unmodified aspect advices. All or almost all of the aspects must be disabled for this kind of attack to succeed.

6. Conclusions and Future Work

We have presented how the combination of a dynamic self-checking mechanism with the generation and remote checking of an idiosyncratic signature allows protecting software running in a remote, potentially hostile environment. We are currently working on a prototype implementation and a more detailed analysis of possible threats is needed.
In future, main effort will be on enhancing prototype development and extensive testing on different case studies.

6. References

[1] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, M. Jakubowski, “Oblivious hashing: Silent Verification of Code Execution”. In Proceedings of 5th international workshop on information hiding (IHW 2002), Noordwijkerhout, The Netherlands, 7–9 October 2002.

[2] C. Collberg, C. Thomborson and D. Low, “Watermarking, Tamper-Proofing, and Obfuscation--Tools for Software Protection,” IEEE Transactions on Software Engineering, vol. 28, 2002.

[3] S. Pearson, B. Balacheff, D. Plaquin, and G. Proudler, “Trusted Computing Platforms: TCPA Technology in Context”, Prentice Hall

[4] E. Valdez and M. Yung, “Software DisEngineering: Program Hiding Architecture and Experiments,” Information Hiding 1999.

[5] S. Kent and R. Atkinson, “Security Architecture for the Internet Protocol” RFC 2401, Nov.1998.

[6] A. Frier, P. Karlton, and P. Kocher, "The SSL 3.0 Protocol," Netscape Communications Corp.,1996.

[7] T. Dierks, C. Allen, “The TLS Protocol Version 1.0,” Internet Engineering Task Force, RFC 2246, Standards Track, 1999.

[8] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, K. Yang, On the (Im)possibility of Obfuscating Programs - CRYPTO 2001

[9] B. Horne, L. Matheson, C. Sheehan, and R. E. Tarjan, Dynamic Self-Checking Techniques for Improved Tamper Resistance. On ACM Workshop on Security and Privacy in Digital Rights Management, 2001.

[10] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3), May 1999.

[11] G C. Necula, "Proof-Carrying Code". On Conf. Proc. of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages", Paris, France, January 1997.

[12] C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of software–based survivability mechanisms. In *IEEE/IFIP International Conference on Dependable Systems and Networks, Goteborg, Sweden*, July 2001.

[13] D. Aucsmith. Tamper resistant software: An implementation. In R.J. Anderson, editor, *Information Hiding, Lecture Notes in Computer Science 1174*. Springer-Verlag, 1996.

[14] C. Collberg and C. Thomborson. Software watermarking: Models and dynamic embeddings. In *Principles of Programming Languages*, San Antonio,USA, January 1999.

[15] M. Baldi, Y. Ofek, M. Young, Idiosyncratic Signatures for Authenticated Execution of Management Code. In Proc. of DSOM 2003.

[16] A. Popovici, G. Alonso, T. Gross, Just in Time Aspects: Efficient Dynamic Weaving for Java. Proc. of 2nd Int. Conf. on Aspect-Oriented Software Development, Boston, USA, March 2003.

[17] AspectJ homepage. On-line at http://eclipse.org/aspectj/