**Author(s):** Falcarin, Paolo; Baldi, Mario; Mazzocchi, Daniele.
**Article title:** Software Tampering Detection using AOP and mobile code
**Year of publication:** 2004
**Citation:** Falcarin, P., Baldi, M., Mazzocchi, D. (2004) “Software Tampering Detection using AOP and mobile code”. In Workshop on AOSD Technology for Application level security (AOSDSEC), Lancaster, UK, March 2004.
**Link to published version:**
http://aosd.net/workshops/aosdsec/2004/AOSDSEC04_Paolo_Falcarin.pdf

# 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 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 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, 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 slows down reverse engineering, such as customization and obfuscation, techniques that prevents 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. 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 sets 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, 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 an entrusting entity, i.e. another computer or as part of some network interface (e.g., firewall, gateway), running in a trusted environment. The TFG is a module that generates a pseudo-random sequence of n-bit tags (idiosyncratic signature) depending on a random seed and on the information to be sent (e.g. using AES [17] algorithm in counter-mode in order to use a stateless communication with the entrusting entity). Moreover a MAC (Message Authentication Code) of the data packet can be generated from a subset of the current idiosyncratic signature.
The n-bit tags (where n is small) are interleaved in the sequence of messages (e.g., inside data packet headers) that are sent from a first computer through the network to a second computer. At the second computer, the validity of the pseudo-random sequence of n-bit tags is checked and verified by the TTC. Sending a valid pseudo-random sequence of n-bit tags verifies that the first computer has used the appropriate software (programs and parameters). Consequently, the second computer accepts and/or forwards only data packets from well-behaved sources.

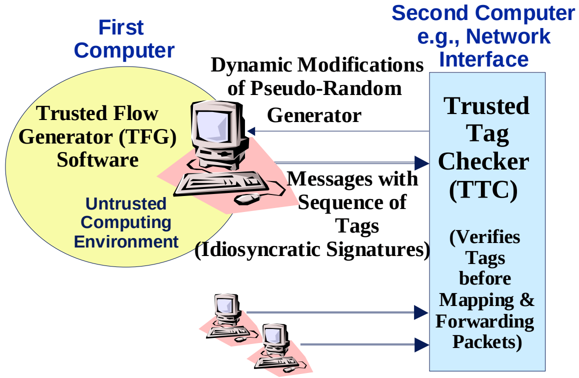
First Computer
Trusted Flow Generator (TFG) Software
Untrusted Computing Environment
Second Computer e.g., Network Interface
Dynamic Modifications of Pseudo-Random Generator
Messages with Sequence of Tags (Idiosyncratic Signatures)
Trusted Tag Checker (TTC)
(Verifies Tags before Mapping & Forwarding Packets)

Figure 1: TrustedFlow architecture

The generator is a dynamic self-checking module and it is the main target of a reverse engineering attack. All existing approaches hide some self-checker in the application running on an un-trusted host. In our approach we consider as variants both the self-checking algorithm and the data used by this algorithm to verify software integrity. Once these features are variants and modularized with AOP they can become dynamically variable in time. Therefore, TTC can timely release new version of TFG, making invalid the former versions. 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 the validity of the attacked mobile code.

## 4. Tampering detection with AOP and mobile code

In this section we analyze possible applications of AOP to detect software tampering, focusing on the implementation of the TFG.

The goal of the proposed methodology is to guarantee “entrusted code” execution in an un-trusted host, allowing detection of the following attacks:

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

The proposed approach is depicted in figure 2.

TrustedFlow is made of a TFG (Trusted Flow Generator) adding bits (i.e. “tag”) to all network packets transmitted to the entrusting entity.

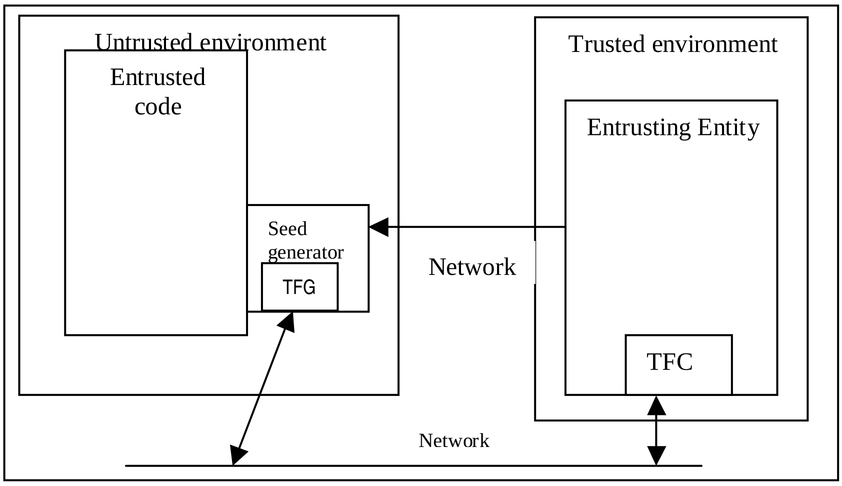

**Figure 2. The TrustedFlow approach with mobile code**

Such tags are generated possibly in a pseudo-random way that depends on the code being executed. The entrusting entity, operating in a trusted environment, includes a TFC (Trusted Flow

Checker) that locally generates the tags and compares them with the ones received from the entrusted code. If the comparison is correct, then the entrusted code has not been tampered.

Seed generator is included in the mobile code: its task is generating the random seed that will be used by TFG as a starting point to generate bits sequences (i.e. tags) inserted in network packets. This tags depends on dynamic checking of entrusted code.

Both seed generator and TFG are executed in an un-trusted host, and they may be enforced with obfuscation techniques to thwart fast reverse-engineering attacks. Disablement of the mobile code makes the application not usable because the TFC module on the trusted counterpart will stop network functionalities of the application.

Reverse-engineering attacks can become more difficult, because the entrusted mobile code is not bundled in the application but it is downloaded at runtime.

Moreover, even supposing de-compilation of the mobile module using ad-hoc packet sniffers and/or memory dump, the periodic change of the mobile code and its own checking algorithms, makes manual reverse engineering difficult, as more as the change period become smaller.

In order to increase the robustness of the above-presented solution two techniques can possibly be used (alternatively or together):

- The mobile module is obfuscated
- The exact output of the mobile module (i.e., the seed and ultimately the tags) depends on a secret key hardwired in such module.

The proposed system can be divided in four sub-problems:

1-*network level*: defining merge algorithm of tags and data generated by the entrusted module; in this way TTC can verify tags (sequence number, handshake, tag flow identifier...). This approach has been investigated in detail in [15].

2-*security level*: defining cryptographic aspects for tag calculation, starting from an initial seed used in generation phase and validation phase.

3 *tag interlocking with code execution*: the choice of input data for seed generator function, must be driven by two ideas: verifying that sensible data are not tampered, and verifying that execution sequence is compliant with original specification used by TTC to perform validation

4 *mobility*: the mobile code can be used to modularize a crosscutting concern like the software tampering checker: the mobility and limited duration of this code becomes the key issue to thwart software cracker.

We analyze mobile code issue in Java environment and we notice that it can be realized with three different technologies: software agents, AOP and dynamic class-loading, dynamic AOP platforms.

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.

So, after a preliminary analysis we think that agents cannot have full control on application code like AOP techniques. In the second approach, static AOP with dynamic class loading can lead to dynamic insertion of code extensions in the application running in an un-trusted host.

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 then these data can be used as input for the seed generator algorithm, contained in the mobile code, and periodically renewed by the counterpart trusted application.

AOP can then insert “hooks” in relevant parts of application code that calls 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; in this case the algorithm, and the related subset, change in time with the mobile code. 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 platform on a standard JVM [16] implies running JVM in debug mode. This means offering a possible weakness to attacker but it has some advantages.

First of all the advantages of static AOP approach are still present, because with dynamic AOP, code structure, attribute values, and methods invocation sequence can be validated with algorithms whose validity lasts for a specified period of time.

Moreover, there is no more presence of “hooks” in 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. Moreover with dynamic AOP, it’s possible to download and withdraw a set of different aspects, each one making some computation to generate the seed, in order to thwart attackers.

Finally the dynamic change of the algorithm and the current data to be checked makes useless a good attack, because of its limited time validity.

Using a dynamic AOP platform or dynamic code downloading means the prototype relies on an external, un-trusted support for mobile code installment and execution. Reliance on such un-trusted support makes the whole system vulnerable, for a limited time.

In future, when dynamic AOP platform will become more secure, then our approach could be improved. However, we do not consider this being a problem for our purposes since our aim is to demonstrate the feasibility of the system through a prototype. A real and robust implementation of the TrustedFlow principle according to the proposed design would include an implementation of the mobile code support (limited to what is needed for the specific issue of handling the mobile module) within the entrusted code itself. In this way, the mobility support will be entrusted through TrustedFlow itself, i.e., the code for the mobility support as well will be interlocked with the TFG.

Finally, using AOP and mobile code for dynamic self-checking can be used to integrate our system with other existing protection approaches. In the following we identify some possible applications.

As Java platform does not allow application to access to the code segment, possible code verification techniques cannot be applied. If the applicaiton is implemented in C++, AOP can be used to calculate a hash of the code segment. 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 are hardware platform independence, 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 the program as it is running, and upon detection to trigger an appropriate response.

Using TrustedFlow protocol any software tampering on the un-trusted host should be remotely detected by TFC, whose response is blocking any further network communication coming from the suspected host.

The two general attacks on a software self-checking mechanism are *discovery* and *disablement*. Methods of discovering such mechanisms, and our approach for preventing or inhibiting these methods, follows.

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 code.

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 using AOP with dynamic class loading, aspect's advice code invocations are identifiable in the code.

Obfuscation of the TFG and the limited time validity of its algorithm can slow down the attack. In this case the time needed to attack 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: not only are there multiple aspects, each one performing a different test (computing a different part of the seed from a different subset of attribute values or code structure), but within each class the testers use different code sequences to do the same job.

*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 TFC 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.

Another possible attack is to identify the mobile code using packet sniffer and/or memory dump: once identified and decompiled the mobile code the attacker can understand mobile code behavior and disable checking activities, but sending the correct information to the TFG.
The time needed to perform such a complex attack is unlikely to succeed without discovery the behavior of all the aspects before their validity expires.

## 6. Conclusions and Future Work

We have designed a dynamic self-checking mechanism suitable to protect client-side software running in a potentially hostile environment. In future, our approach could be used in conjunction with other tamper-resistance techniques, like static copy-specific software customization. Directions for future research include developing of distributed hash functions in different aspects, obfuscations for the mobile code, and adding security features to a dynamic AOP platform.

## 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] Advanced Encryption Standard specification. On-line at http://www.nist.gov/aes