INTEGRITY CHECKING IN REMOTE COMPUTATION

Paolo Falcarin[1], Riccardo Scandariato[1], Mario Baldi[1], Yoram Ofek[2]
[1] Politecnico di Torino, Dipartimento di Automatica e Informatica
[2] Università di Trento, Dipartimento di Informatica e Telecomunicazioni
{Paolo.Falcarin,Riccardo.Scandariato,Mario.Baldi}@polito.it, Ofek@dit.unitn.it

How can a client-side application be entrusted albeit running inside an un-trusted environment? Within an un-trusted environment a possibly malicious user has complete access to system resources and tools in order to tamper with the application code. Under those assumptions, the server-side application needs a way to continuously ascertain that the client code has not been altered prior to and during execution, i.e., the server is required to continuously entrust the client. To address this problem, we propose a novel approach based on the client-side generation of an execution signature, which is remotely checked by the server, wherein affirmative checking ensure the authenticity of the client-side software. The method proposed is applicable to remote computation, in general, and has the potential to solve some of the central trust problems in GRID computing.

1. Introduction

The level of security and trust offered by present software systems is not satisfactory, especially considering the strategic role such systems play in modern economy. Inevitably, the damage to the community caused by insecure software artifacts has become substantial, as demonstrated by various incidents in the recent past. The outcome of above-mentioned security threats causes both monetary losses and a weakened trust in key software systems.

Many circumstances do exist in which it is desirable to protect a software module 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 security 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 for far-flung penetration, intrusion, and (distributed) denial of service.

This work was supported in part by funds from the European Commission (contract N° 002807)

In particular, this work has been carried out in the context of a research project aiming at providing an answer to the following question: *How can a client application be entrusted albeit running inside an un-trusted environment?*

With the term “un-trusted environment” we mean a computing base (e.g., networked computers and mobile devices) in which a possibly malicious user has complete (conceivably physical) access to system resources (e.g., memory, disks, and so on) and tools (e.g., debuggers, disassemblers, etc.) in order to tamper with the application code. In the scope of this paper, an application is deemed *trusted* whenever its executed code has not been altered. Getting the above-mentioned applications to be trusted in such a hostile environment is a challenging, yet largely un-addressed issue. TrustedFlow [5] is our software solution methodology to specifically tackle the problem of remotely authenticating software during execution in order to assure that it is not altered prior to and during execution.

In our approach, the answer to the above-mentioned problem is achieved by continuously emanating a flow of idiosyncratic tags from clients to server. The tags are used as evidence to the server that the client code emanating them is authentic. The tags flow is generated by a secret function hidden in the to-be-entrusted software (*entrusted entity*) and whose execution is subordinated to the genuine execution of the client software itself. The flow of tags is validated at a remote component (*entrusting entity*) that is executed in a trusted environment. This generation and validation method of idiosyncratic tags is called TrustedFlow.

Remote entrusting of applications is also an emerging requirement for distributed computing architectures (e.g.: GRID [7]) where several computation units are deployed on a large number of remote untrusted platforms. Thus, malicious tampering with a single computation unit can cause data errors which compromise the entire distributed computation: this kind of attack can waste the work of days of GRID computing.

An implementation of TrustedFlow, discussed in this paper, is based on the deployment and nonstop replacement of dynamic integrity checking modules. In general, traditional integrity self-checking has the problem of how to make sure that the self-check is performed. In this implementation, the remote verification provides a way of verifying that the check has been duly preformed. The proposed solution can be deployed in scenarios in which access to servers should be allowed to original client software. Key examples of existing applications include: Yahoo service access (Yahoo advanced services are available only to users deploying Yahoo’s client), gambling servers, and on-line exam tests. Along these lines, a chat system, composed of a server acting as entrusting entity and a client acting as entrusted entity, has been developed as a proof of concept.

Next section describes the operating principles of TrustedFlow.

2. Basic TrustedFlow Principles

The TrustedFlow solution has two basic principles.

1. **Interlocking** describes the combination of the entrusted software (original function) with an idiosyncratic tag generator, which generates an unpredictable flow of tags. Interlocking aims at assuring that the idiosyncratic tag generation is bound in an inseparable manner to the actual functional code and than can be used as evidence of integrity to the server.

2. **Hiding** is the general term describing the necessary countermeasures to ensure that reverse engineering the tag generator and breaking the interlocking is difficult enough so that it becomes practically infeasible.

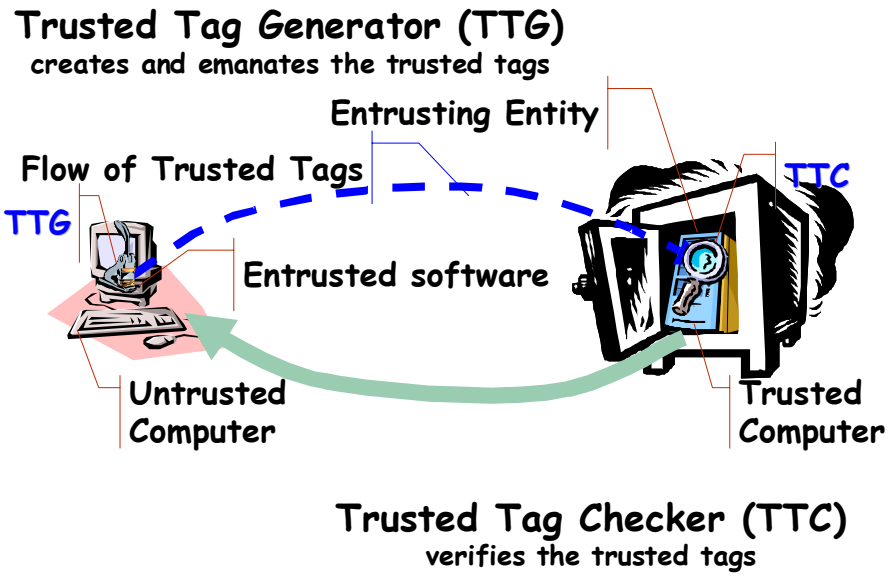

Figure 1: TrustedFlow system architecture

During runtime, the following two components, depicted in Figure 1, enact the TrustedFlow method for remote authentication of software execution.

1. A **Trusted Tag Generator** (**TTG**) within the entrusted software constantly generates a flow of tags, constituting the continuous idiosyncratic signature (that cannot be forged) of the entrusted software's execution, and binds them to traffic generated by the entrusted software.

2. A **Trusted Tag Checker** (**TTC**) within an entrusting entity authenticates the entrusted software by verifying the flow of tags that forms the continuous idiosyncratic signature emanated by the entrusted software.

For increased robustness, the algorithm used to generate the tags should not require a strong synchronization between TTC and TTG. For instance, current implementation uses a block cipher in counter mode and includes the counter value among the data transmitted between the TTC and the TTG. See Section 4 for further details.

Moreover, cryptographic functions can be employed to bind tags to transmitted data. For example, a message authentication code (MAC) of the data unit including the related tag can be attached by the TTG to protect against the tampering with the data associated to a valid tag. Alternatively, a MAC calculation can be part of the tag generation algorithm.

3. Mobility to Build Trust

Our solution to the implementation of TrustedFlow realizes the *interlocking* principle by means of a *code integrity checker* [1] that continuously monitors code execution and controls the idiosyncratic tag generator. In case the code integrity checker returns a negative result, generation of the idiosyncratic tag flow is inhibited. Dynamic checking [4], in which the program is verified repeatedly at runtime, is preferred rather than static integrity checking, in which the program integrity is (self) checked only once, during start-up.

To counter reverse engineering, hence implementing the hiding principle, current integrity checking techniques mostly rely on co-bundled code checkers whose position is hidden in the application and whose behavior is obfuscated or complex to understand [2,3].

However, we observe that any technique involving a checker that is permanently co-bundled within the application is not robust enough. Indeed, the checker can be eventually identified and inhibited by an attacker with enough knowledge, time, and reverse engineering tools. Under such conditions, there are no guarantees that a remote client is actually undergoing to all the checks he is supposed to.

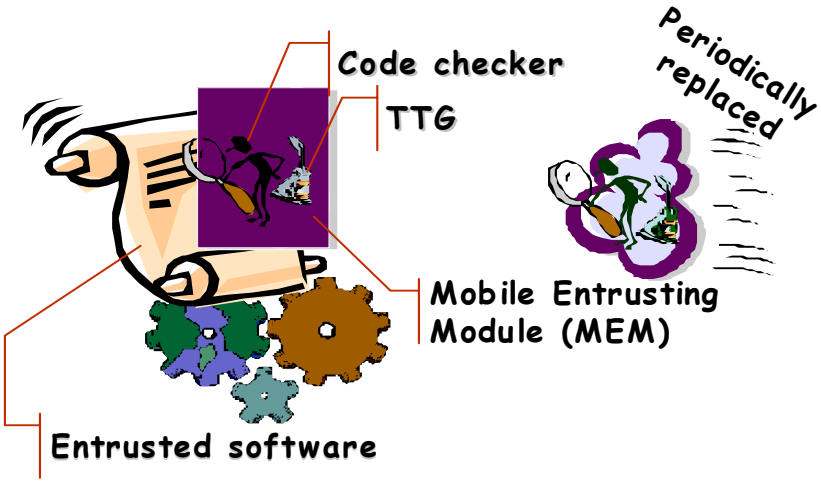

Figure 2: Mobility for the hiding principle

Our implementation of the *hiding* principle is based on *mobility*, possibly combined with mild obfuscation. As graphically represented in Figure 2, the checker is bundled as an independent module, called **Mobile Entrusting Module (MEM)**, which is sent to the client at startup and can be updated dynamically at any time. The update rate can be tuned according to the security

requirements of the target application domains (from minutes to days). Nonetheless, anti-tampering techniques, such as obfuscation, are still a valuable addendum in order to make it even harder for a rogue user to hack the checker code. A checker that is both mobile and obfuscated gives an additional degree of freedom to customize the security level: the stronger obfuscation, the lower the update rate can be, and vice versa.

In summary, when compared to typical integrity checkers, our approach extends their power in two ways:

1. It adds remote verification that checking has been actually performed.

2. It supports the continuous replacement of the checking code.

4. Proof of Concept

To show the applicability of the proposed approach we developed a prototype implementation of the remote code verification architecture described in Section 3. As a "toy" example, we used a chat application composed by a Java client and a chat server relaying user messages to chat participants. The client side has been left unmodified. Therefore, once the chat client program is released and distributed to users, an attacker cannot get any clue about the integrity strategy the server will adopt.

Concerning the server program, it has been extended to integrate it with the TrustedFlow server module (TTC, in Section 2).

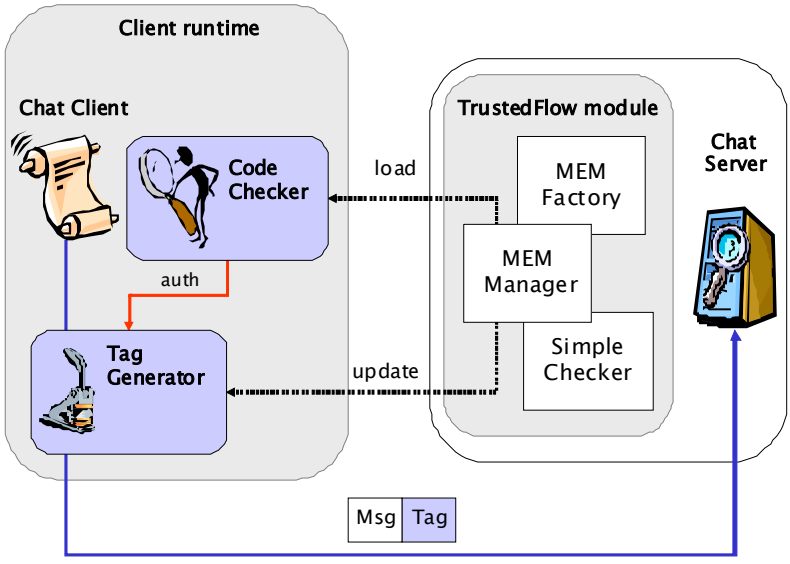

Figure 3. Prototype architecture

As shown in Figure 3 (right-hand side), main components of the TrustedFlow modules are the `MEM Manager`, the `MEM Factory` and the `Simple Checker`. MEM Manager provides the Chat Server with the interface to access

TrustedFlow functionalities, namely the registration of a new client and the verification of tagged messages. It is supported by the Simple Checker, which validates the tags carried by user messages, and by the MEM Factory, which dynamically generates the code of the to-be-deployed modules.

In the prototype, the MEM is modularized into two components that can be independently replaced or modified, making future experimentation and enhancements easier. As shown in left-hand side of Figure 3, when a new client comes in, the TrustedFlow module loads the two components in the execution environment of the upcoming client. They are the `Code Checker`, which behaves as a watchdog for the client program and looks for integrity breaches, and the `Tag Generator` (TTG in Section 2), which seamlessly append a tag to each user message.

In our prototype the Code Checker defines a “sandbox”, i.e. a set of methods and libraries the client application is allowed to use. For instance, an application profiler can be used to automatically extract the list of called methods by the client code. Such list is embedded in the Code Checker and it used to enforce controls over the sandbox boundaries. As soon as the client code trespass the sandbox limits, the Tag Generator is informed. The Tag Generator intercepts network transmissions to the chat server in order to obliviously insert tags in the data sent out by the client application. Valid tags are generated until the Code Checker certifies the client code is genuine. When the generator is informed of a sandbox violation, it generates illegal tags that will eventually alert the TTC.

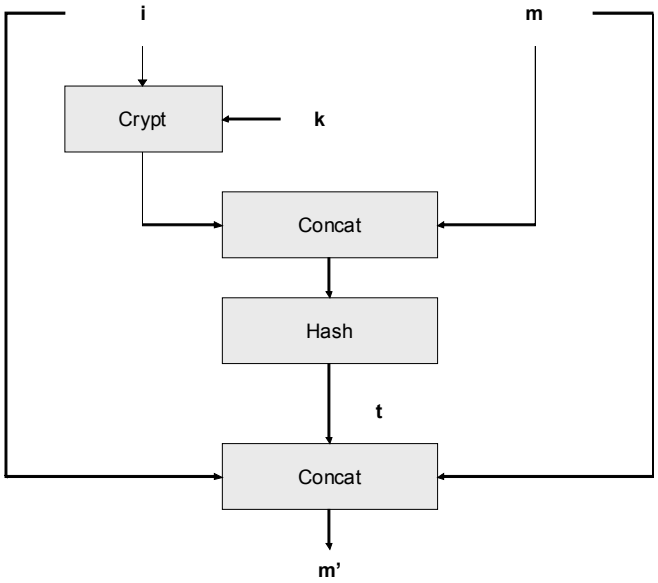

Figure 4. Tagged message generation

Tags are generated according to the algorithm depicted in Figure 4. Each Tag Generator has a counter that is incremented each time a new message is sent out. The Tag Generator also shares a symmetric secret encryption key with the TTC. Each client has a different key. Furthermore, each key can be changed at any time by updating the corresponding Tag Generator. Using the secret key

(*k*), the Tag Generator encrypts the current counter value (*i*) with the AES block cipher [66].

The resulting block is concatenated with the plain user message (*m*) and then hashed[2] to produce the tag (*t*). The resulting tagged message (*m'*) is made by the plain message, the counter value, and the calculated tag.

While the Code Checker is pushed at runtime and never replaced (for simplicity), the TrustedFlow module automatically updates each Tag Generator whenever the corresponding user sent out a given amount of messages or the ageing timer expired (whichever comes first).

5. Conclusions

This paper presented an innovative approach to deal with remote verification or entrusting of client-side application code. The proposed solution broadens classic integrity self-checking techniques by both making them more robust (thanks to mobility), and providing the additional feature of continuous verification of integrity by a remote (trusted) server. As a positive outcome, the strategy adopted by the checker is not inferable through static code observation, i.e. the checker arrives at run-time and it will change periodically during execution. During runtime, the attacker task is made even harder due to continuous replacement of the checking strategy.

The paper also presented a workable prototype implementation of the proposed approach for a chat application. The current prototype was meant as a demonstrator and, thus, it suffers some limitations. Mainly, the integrity check technique is quite simplistic as the Java platform does not allow application access to the code segment, and, hence, code verification techniques cannot be applied.

Our code checker adopts an anomaly detection technique and can immediately spot when program is traversing the boundaries of the sandbox. However, this does not strictly guarantee that the original program has not been altered. More sophisticated approaches are based on low-level inspection of executed bytecode, e.g. as to compare the hashed running bytecode with a trusted copy.

The above limitation can be overcomed if the application were implemented in C++. The mobile module could calculate a keyed hash of the code segment at runtime. Such technique 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 in detecting an attack where the program is modified temporarily and then restored as soon as the malicious behavior took place.

6. References

[1] Falcarin, P., Baldi, M., Mazzocchi, D., 2004, *Software Tampering detection using AOP and mobile code*, Workshop on AOSD Technology for Application level security (AOSDSEC), co-located with AOSD 2004, Lancaster, UK.

[2] The algorithm uses a low-collision cryptographic hash function.

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

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

[4] Chen, Y., Venkatesan, R., Cary, M., Pang, R., Sinha, S., and Jakubowski, M., 2002, *Oblivious hashing: Silent Verification of Code Execution*, Proceedings of 5th international workshop on information hiding (IHW 2002), Noordwijkerhout, The Netherlands.

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

[6] Daemen, J., and Rijmen, V., 2000, *The Block Cipher Rijndael*, Smart Card Research and Applications, LNCS 1820*, J.-J. Quisquater and B. Schneier Eds.,* Springer-Verlag.

[7] The GRID web-site. On-line at http://www.grid.org