# Application-oriented trust in distributed computing

Riccardo Scandariato
Katholieke Universiteit Leuven
Belgium
first.last@cs.kuleuven.be

Yoram Ofek
Università di Trento
Italy
last@dit.unitn.it

Paolo Falcarin, Mario Baldi
Politecnico di Torino
Italy
first.last@polito.it

## Abstract

*Preserving integrity of applications being executed in remote machines is an open problem. Integrity requires that application code is not tampered with, prior to or during execution, by a rogue user or a malicious software agent. This paper presents a methodology to enforce run-time integrity of application code by means of an integrity-preserving software component that is combined with the application. The software component is a trusted logic that can be replaced continuously from a remote location during run-time. For added assurance, the software component produces continuous sequence of proofs of its proper operation that are verified remotely. The paper discusses the general principles of operation of the above-mentioned methodology and presents the results of experimentation in the context of client-server applications.*

## 1. Introduction

Consequently to the convergence of networking and computing, software integrity is becoming a growing problem. More and more often, applications are built out of several distributed components, possibly from different administrative domains, which collaborate through the network (e.g., web services). As a result, various trustworthiness issues emerge because of intentional misbehavior. Furthermore, machines connected to the Internet are exposed to malware (viruses and worms) and cannot be considered trustworthy, notwithstanding the owner is properly behaving. Therefore, from a server perspective, there is high risk in accepting service requests from either 3rd-party business components or end clients. First, the server can be attacked by its clients through legitimate, i.e., unfiltered communication channels. Second, the service can be exploited as a vector to infect other parties. The recent proliferation of bots and viruses spreading through popular instant messengers is a clear testimony.

The general approach we propose is called *remote entrusting* and uses trusted entities in the network (e.g., servers) in order to entrust selected software elements (e.g., applications) in otherwise untrusted machines across the network, assuring their run-time functionality. Note that, in this paper, the notion of trustworthiness is related to integrity of code, whereas data protection is not considered, although ongoing work is addressing this aspect as well.

Detection of run-time software changes (e.g., to circumvent the license, or because of virus infestation) across the network is hard, since the entrusting entity cannot directly observe the software executing on the remote end. In order to solve the problem, the entrusting entity should receive some "proofs" that guarantee the run-time integrity of the software. The proofs are represented by tags that are emanated from selected parts of the application on the remote environment, and they provide the identity of the running software. Thereby, they enable the entrusting entity (server) to entrust the software running on the opposite end (application). More specifically, this work addresses two research challenges: (1) how to enhance applications by binding a proofs-generating module that continuously emanates secure proofs, and (2) how to periodically replace the module during run-time in order to cap the time at hand for both tampering attempts and tag forgeries.

Another related challenge, which is part of remote entrusting but outside the scope of this paper, is that reverse engineering of the proofs-generating module shall be hard to perform [3]. Further, proposing a code integrity checking scheme is not a goal for this paper. A simple scheme is used in our prototype. However more advanced code checkers can be adopted from the extensive state of the art (e.g., [2]).

The paper is organized as follows. Section 2 introduces the principles of operation of the proposed methodology. Section 3 presents the prototype implementation, whose performance is experimentally assessed in Section 4. In Section 5, possible threats and countermeasures are discussed. Finally, Section 6 compares the remote entrusting to related work, while Section 7 presents the concluding remarks.

## 2. Remote entrusting

The solution to remote entrusting that we propose in this paper in intended for use in a setting where there is an exchange of data from the to-be-trusted entity and the entrusting side. That is, there is a data flow going the opposite direction of the trust relationship. Instant Messaging (IM) services are a typical example of such architecture. For instance, the server can be interested in banning communications coming from bot-infected clients, in order to block the spreading of malware via its channels. Furthermore, the service provider could be interested in forcing the adoption of a specific client because it may incorporate usage policies (e.g., limiting the message rate) to contain malign behavior, like spamming. In this respect, the IM server represents the entrusting side, which receives and relays messages from clients (the to-be-trusted entities). The above-mentioned scenario is used hereafter to illustrate the remote entrusting method in practical and more comprehensible terms. In the rest of the paper, some terms are be used according to a specific meaning: *trust* means reliance on integrity of code of an entity, e.g., an application, *trustworthiness* means deserving trust, and *entrusting* is the act of according trustworthiness.

The proposed solution is designed around some fundamental principles:

1. Root of trust at remote location – The basic working assumption when dealing with trustworthiness is that some elements in the system are implicitly entrusted and the whole infrastructure founds on them. In this work, the root of trust is placed in a remote site across the network.

2. Continuous replacement – The root of trust deploys a software beach-head on the untrusted host that is responsible for preserving the integrity of the to-be-entrusted application. The beach-head can be replaced at any time by the root of trust.

3. Trustworthiness during run-time – This work introduces a protocol providing application-oriented trustworthiness that is refreshed during run-time. This introduces an on-line and proactive method to avoid trustworthiness violations and application damage.

**Trustworthiness establishment protocol.** Proofs are produced by the trusted replaceable beach-head on the client side and they are comprised of tags attached to data exchanged from client to server. Each tag contains the outcome of a cryptographic algorithm tying together the exchanged data and some secret information, which is hidden in the beach-head itself. For instance, the secret information could be a symmetric encryption key, shared between the beach-head and the root of trust. As long as the application under surveillance is in proper shape, the beach-head keeps producing tags correctly. Upon reception of a tagged message, the root of trust validates the tags and hence certifies the trustworthiness of of the application.

Note that the methodology that is presented here can be extended to other important application domains. For instance, remote entrusting could be used in Digital Right Management to enforce intellectual property rights. That is, the root of trust could be configured to act as a license manager. As a bonus, the license manager would also be able to prevent software corruption by malicious agents. Furthermore, the client-server scenario could be applied the way around, i.e., to entrust on-line services from the client perspective. For instance, a home-banking client could be very interested in verifying integrity of the server application it is connecting to, e.g., in order to avoid connection to a server that has been subverted or even spoofed.

**Replacement.** Protecting the trustworthiness of replaceable beach-head, both in terms of performed functionality and secrecy of information contained within, is key. In this respect, obfuscation have been proposed in the past to hinder adversaries, but motivated and skilled individuals could eventually succeed in tampering with the application code. Our objective is to make it intractable, within a well-defined period of time, for a malicious agent to modify selected components of application without being detected by the root of trust. To this aim the client application is supplemented with a software beach-head that both controls the application integrity and generates tags. The beach-head can be replaced by the root of trust at any time during the application run-time. The replacement beach-heads can contain new secret information (e.g., a fresh key), new integrity checking strategies, new algorithms for tag generation, and so on. The interval between two subsequent replacements is the time window that is left to the adversary to break the integrity of the software beach-head and, hence, of the application itself.

Replacement offers an additional degree of freedom to play with in protecting the application. The complexity of reverse-engineering the combined unit formed by the client application and the replaceable beach-head can be directly translated to the time an adversary needs in order to break it (trustworthiness window). As far as the beach-head is replaced at a pace that is faster than the inverse of the trustworthiness window, the combined unit can be safely considered not tampered. Clearly, understanding the precise reverse-engineering complexity of tamper-resistant software,e.g., software employing obfuscation [3] and white-box cryptography [7], is not a trivial job and the research field is still open to further study. Nonetheless, it is also possible to trade off between the complexity and

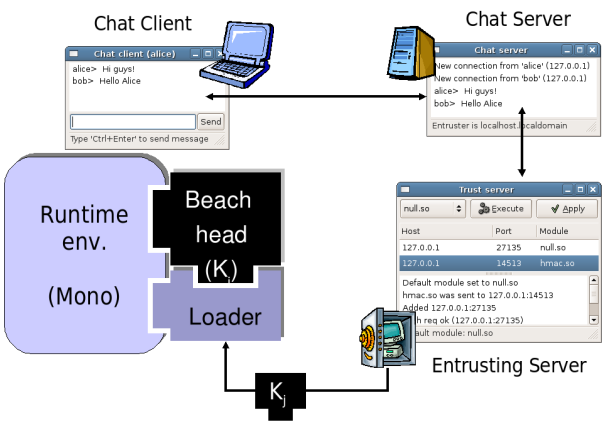



Figure 1: Prototype architecture

- The loader contacts the entrusting server, registers, and downloads the first integrity-preserving beach-head.
- The beach-head is plugged into the run-time, to be invoked on each method execution.
- The loader opens a server socket and waits for replacement commands from entrusting server.
- The application is actually started.
- At any time, the loader can receive a new beach-head from entrusting server by means of a replacement command. Of course, a new beach-head must be accepted only from an authenticated entrusting server, e.g., by means of an authenticated communication channel, or through signed beach-heads.

the replacement rate. Hence, in case of sufficiently high replacement rate, the adoption of above-mentioned techniques could be taken down to the very minimum, if not avoided at all. This is good result if we take into consideration performance degradation due to anti-tampering techniques.

The performance overhead introduced by the replacement (and hence the limits to the replacement rates) is investigated in the next section.

## 3. Prototype

To show the feasibility of the proposed approach, we implemented the remote entrusting method for a chat-like messaging application. The prototype is written in C# programming language and runs on the Mono run-time. The code is available online as open source [1]. Figure 1 depicts the prototype architecture.

### 3.1. Client side

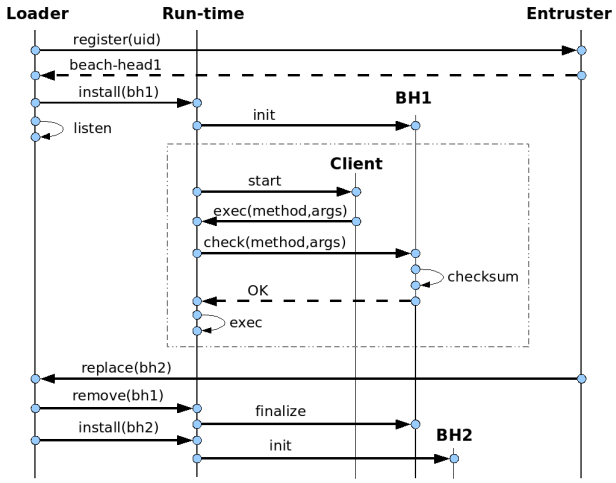
**Chat client.** The chat client is a C# object-oriented application representing the to-be-entrusted software. All methods of all application classes are seamlessly intercepted by the entrusting infrastructure (loader and beach-head) as explained hereafter.

**Loader.** The loader is a small user-space software component permanently installed in client environment. The loader functionality is very limited, and it needs not to be trustworthy. When the chat client is started, the loader is plugged in Mono run-time. Then, the loader executes the following communication protocol, also depicted in Figure 2a:

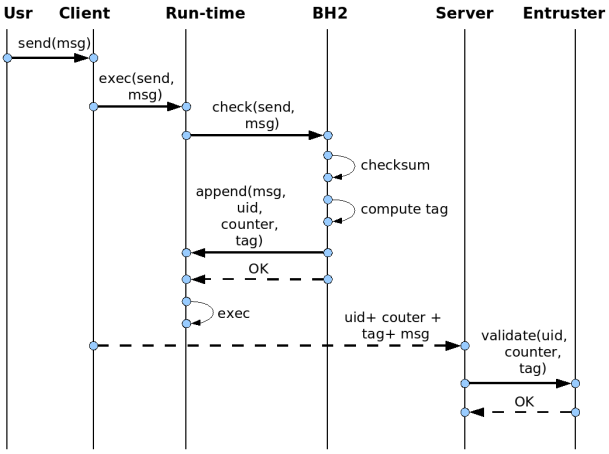
**Beach-head.** The replaceable beach-head shares a symmetric key with the entrusting server, and each beach-head instance has a different key. The beach-head is invoked by the Mono run-time before any application method is executed. The beach-head analyzes the code that the Mono run-time is going to interpret to execute the method. Hence, the integrity of code is constantly check-pointed all over the application run-time. The beach-head calculates a keyed hash of the code and compares the computed checksum with the value that was pre-computed at the server side for a program copy that is known to be genuine. The pre-computed value is embedded in the beach-head itself. If the hash does not match the genuine copy, the key is randomly altered, so that valid tags will no longer be produced in the future.

**Run-time.** The Mono interpreter does not have a built-in functionality to intercept the methods conveniently. Hence, we extended the Mono code-base to serve our purposes. The changes are limited to about 300 LOC.

**Tag generation.** When the client sends a message, the corresponding method call is intercepted to attach the tag, as depicted in the boxed part of Figure 2a. If at any time in the past the beach-head failed at authenticating the code of some method, the encryption key used to generate tags has been irrevocably invalidated and valid tags cannot be generated. Supposing that all previous checks were successful, a valid tag is generated by the beach-head according to the algorithm of Figure 3. Note that tagged messages are sequenced for two reasons. First, since the message counter is included in the message, loose synchronization is requested between the beach-head and the entrusting server. Second, this allows the entrusting server to defend against replay attacks.




(a) Initialization and check-summing




(b) Tag generation and validation

Figure 2: Sequence diagram

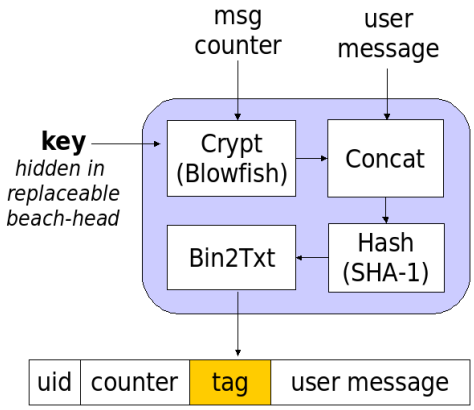



Figure 3: Tag generation schema

### 3.2. Server side

**Chat server.** The chat server is a graphical application implementing the single chat-room logic. The server relays messages received by registered users to all connected clients. As shown in the bottom part of Figure 2b, before a message is accepted it is forwarded to the entrusting server in order to verify the validity of the attached tag.

**Entrusting server.** The entrusting server receives tagged messages, identifies the message originator by means of the unique source ID included in the message itself, recovers the symmetric encryption key associated to that specific client, and computes the tag generation algorithm on the message data. If the message tag corresponds to the calculated tag, the message is valid and a positive acknowledgment is sent back to the chat server. Otherwise, the chat server will be alerted and the message ignored. Additionally, at any time, the entrusting server can replace the current beach-head in a client by issuing a command.

## 4. Performance Analysis

Performance analysis was extensively carried out for our C# prototype. Tests were performed on a machine with a Pentium IV processor, clock speed of 3.2 GHz, 2048 KB cache, and 1 GB of RAM. The machine was running Linux, kernel version 2.6.12-10. We used the Mono run-time version 1.1.15 (patched). Measures where obtained by means of the application profiler included in the Mono run-time.

The application was configured so that the server, the client, and the entrusting server resided on the same test machine. All communication took place locally, in order to purge the (variable) influence of transmission delays from collected data. During each test the chat client was started, it connected to the server, sending its nickname in order to register with the chat room, and, after the GUI had finished initializing, the client was terminated automatically. Each experiment was repeated 10 times.

We run four different experiments to test the client side:

- Measure of start-up overhead (indirectly sizing the overall overhead due to entrusting infrastructure).
- Measure of methods checksum computation overhead.
- Measure of combined overhead due to both checksum computation and tag generation.

| | | No beach-head | Null beach-head | Crypto beach-head | |
|---|---|---|---|---|---|
| Start-up | Average | 534.774 | 1041.951 | 1103.471 | ms |
| | Std dev | 2.999 | 30.332 | 48.733 | ms |
| | Degradation | | +94.84% | +106.34% | |
| Integrity checking | Average | 4.719 | 5.229 | 5.353 | ms |
| | Std dev | 0.030 | 0.026 | 0.025 | ms |
| | Degradation | | +10.81% | +13.44% | |
| Tag generation | Average | 1.763 | 1.870 | 2.118 | ms |
| | Std dev | 0.014 | 0.016 | 0.013 | ms |
| | Degradation | | +6.09% | +20.16% | |

Table 1: Summary of performance overhead

- Measure of disruption of service during replacements.

### 4.1. Start-up overhead

This experiment is aimed at measuring the overhead due to the entrusting infrastructure on the start-up time of the client. First, we run 10 tests without installing any loader on the client (base line). Then, we repeated the experiment with the presence of the entrusting infrastructure. In this case, the beach-head intercepts the method calls but do not execute any integrity check (null beach-head). Of course, in this second case there is an overhead due to the following causes: (1) during startup the initial beach-head must be downloaded, initialized, and installed by the loader, and (2) the beach-head is invoked each time any C# method is entered (including sub-calls to virtual machine methods). The second cause is largely predominant. Finally, we executed a third series of tests with an integrity checking beach-head calculating cryptographic checksums with a 128 bits key (crypto beach-head). Note that in this case, each method call is intercepted and checksum computed.

As shown in the first row of Table 1, in case of no entrusting, the test executed in about 534.8 ms on average. When entrusting infrastructure was enabled (i.e., the null beach-head is deployed in the client run-time), the performance deteriorated of about 95%. Finally, an additional 11% degradation is to be added when the real beach-head is deployed. In conclusion, the main performance degradation is due to the method interception mechanism in the Mono platform.

### 4.2. Integrity checking overhead

This experiment aims at highlighting the computational load introduced by the checksum calculation. As before, we run three series of tests: (1) the plain client, (2) the client with a null beach-head, and (3) the client with the crypto beach-head. Among all application methods, we monitored the `output()` method, a tiny method that performs console output only. The results are shown in the second row of Table 1.

We are interested in understanding the extra-overhead due to checksum computation (crypto beach-head), compared to the existing overhead of the interception mechanism (null beach-head). According to our experiment, the performance deteriorates of only 2.6%, which is an important result.

### 4.3. Tag generation overhead

This experiment monitors the `Write()` method, i.e., the method sending messages to the server over the network. In this case, the beach-head, not only checks the integrity of the send method itself, but also generates a tag and attaches it to the user message (in a transparent way). Similarly to previous experiments, we run three series of tests: (1) the plain client, (2) the client with a null beach-head, and (3) the client with the crypto beach-head.

The results are shown in the third row of Table 1. As before, we compared the performance values of last two columns and we observed a performance degradation of about 14%.

### 4.4. Replacement time and replacement rate

When the beach-head is replaced, the client application suffers a short disruption of service. Replacement time is measured as the interval between the withdrawal of the old beach-head and the installation of the new one. Time needed to receive the replacement beach-head is not taken into account, because it does not cause any disruption, as the old beach-head is still in place. Replacement of the crypto beach-head takes 2.675 ms on average (standard deviation is 0.195 ms).

In conclusion, the interval during which the client is not responding because a replacement is tacking place is negligible. More importantly, since the size of the beach-head is tiny (134 KB) and the replacement time very small, the replacement rate (from the server perspective) can scale up to few seconds, if the client network bandwidth is large enough.

## 5. Discussion

The approach that we presented and implemented can be subject to a number of threats, which are briefly discussed here.

**Attempt to remove the beach-head.** This threat is represented by a malicious agent trying to separate the client from the beach-head. However, the beach-head is a functional part for the client, i.e., the beach-head is not only responsible for checking the application trustworthiness (non-functional part). The beach-head also includes the generation of the tags, which is an essential functionality in order to be able to inter-operate with the server. Moreover, the beach-head could be extended in order to include additional functional parts of the application itself. Therefore, attacks aiming at merely disabling or bypassing the beach-head are not effective.

**Attempt to forge messages.** This threat is represented by a malicious agent trying to send messages that have not been generated by a trustworthy client. For instance, the malicious agent could run an altered copy of the client in parallel to the trusted one and reuse the tags produced by the latter. Similarly, the malicious agent could be trying to spread some malware via forged messages containing a malicious payload. This attack is ineffective because each tag is cryptographically bounded to the content of the message it is appended to. Therefore, is not valid when used for other bogus messages.

**Attempt to substitute the beach-head with a fake one.** This threat is represented by a malicious agent trying to extract the secret key from the beach-head in order to replace the latter with a mock-up, which produces valid tags but does not check the trustworthiness. This threat is countered by the combined action of the frequent beach-head replacement (as showed in the prototype) and the beach-head obfuscation (not implemented in the prototype).

**Attempt to trick the beach-head.** This threat is represented by a malicious agent trying to prevent the beach-head module from validating the code integrity of the client, e.g., by tampering with the OS system calls or the virtual machine libraries. For instance, in our prototype, the run-time could be modified in order to provide the beach-head with authentic code from a cache, even though the actually interpreted code is indeed altered. This threat is real and requires further work. A possible improvement to address this issue is to let the beach-head extend its reach, e.g., by checking the trustworthiness of selected parts of the environment, e.g., the run-time the client is running on.

## 6. Related work

The problem of executing software in a trustworthy computing environment has recently gained considerable attention. The literature can be divided into either hardware-based or software-based solutions.

The Trusted Computing Group is defining a set of standards to address the problem of executing software in a trustworthy computing environment from a hardware perspective. The approaches under this umbrella [7, 9] build on top of a tamper-proof hardware component, e.g., the Trusted Platform Module, which is situated locally on the motherboard. Hardware components cannot be replaced in case of design glitches. Moreover, trustworthiness covers the machine as a whole (including BIOS and OS) and cannot be granted at a fine-grain level, e.g., for selected applications. Finally, the integrity verification method is off-line and reactive, i.e., after the fact.

Pioneer [12] is a software-based system that relies on a verification function running on the client as an operating system primitive, and an attestation server. The verification function acts as a software counterpart of the tamper-resistant chip of Trusted Computing. The function is designed in such a way that an attempt to tamper with it results in slower computation. However, some assumptions in this approach are not realistic in most practical cases, and the approach is not applicable on an Internet scale. For instance, verification function does not work on multi-processor or multi-core computer architectures. Furthermore, the verification function and the attestation server must be wire-connected and the client must not communicate with anybody else during the challenge phase. Otherwise, in a LAN setting, special configuration is required for the switches.

Kennel et al. present an interesting software-based approach to verify the trustworthiness of an OS kernel [6]. The OS trustworthiness is established by means of computation of cryptographic hashes over selected memory portions (via a challenge set by the attestation authority). Further, assumptions on slower computation of emulators are used to detect kernels that are running inside emulators. The work could be integrated in our methodology in order to entrust the platform together with selected applications. Garay et al. present a similar approach where a trusted challenger

sends a challenge to the potentially corrupted responder. The challenge is an executable program that can execute any function on the responder. To prove its integrity, the responder must compute the challenge within the time bounds set by the challenger.

Hohl presents an interesting approach in the area of protection for software agents [4]. The central idea of the work is to generate an executable agent from a given agent specification which cannot be attacked by read or manipulation attacks, i.e., a black-box. Further, the black-box is associated with an expiration date limiting its validity over time. However, no practical implementation is investigated for the proposed approach.

Another relevant area of related work is represented by techniques to harden mobile agents [8, 10, 11], which can be applied to the proofs-generating module. Further, our periodic replacement strategy relies on the assumption that tampering attempts can be made hard if the lifetime of the proofs-generating module is limited. This approach has similarities with software aging, where new updates of a program are frequently distributed [5].

## 7. Conclusions

Computing entities continuously interact across the network, and, as a result, critical trustworthiness problems are emerging. According to the definition adopted in the context of this work, an application is deemed trustworthy if, and only if, its functionality has not been altered prior to or during execution. That is, the solution we propose puts particular accent on remotely verifiable guaranties that the software base running on clients is genuine. This work presented the remote entrusting methodology: a software-based, low-cost solution that can be employed either to detect and contain malicious software agents (e.g., virus-born bots) trying to sneak in and compromise a network-connected computing environment, or to counter intentional tampering attempts with software, e.g., to obtain illegal access to services, to break the license, to circumvent hard-coded policies, and so on.

## References


[1] Prototype code available online. `http://www.cs.kuleuven.be/~riccardo/index.php?page=TrustedFlow`.

[2] D. Aucsmith. Tamper resistant software: an implementation. In *International Workshop on Information Hiding*, Cambridge, UK, May 1996.

[3] C. S. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. *IEEE Transactions on Software Engineering*, 28(8), August 2002.

[4] F. Hohl. Time limited blackbox security: Protecting mobile agents from malicious hosts. In *Mobile Agents and Security*. Springer-Verlag, 1998.

[5] M. Jakobsson and M. Reiter. Discouraging software piracy using software aging. In *ACM Workshop on Digital Rights Management*, Philadelphia, PA, November 2001.

[6] R. Kennell and L. H. Jamieson. Establishing the genuinity of remote computer systems. In *USENIX Security Symposium*, Washington, DC, August 2003.

[7] P. C. V. Oorschot. Revisiting software protection. In *International Conference on Information Security*, Bristol, UK, October 2003.

[8] A. D. Rubin and D. E. Geer. Mobile code and security. *IEEE Internet Computing*, 2(6), November 1998.

[9] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a TCG-based integrity measurement architecture. In *USENIX Security Symposium*, San Diego, CA, August 2004.

[10] T. Sander and C. F. Tschudin. Protecting mobile agents against malicious hosts. In *Mobile Agents and Security*. Springer-Verlag, 1998.

[11] T. Sander and C. F. Tschudin. Towards mobile cryptography. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 1998.

[12] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *ACM Symposium on Operating Systems Principles*, Brighton, UK, October 2005.