Remote Entrusting by Run-Time Software Authentication

Mariano Ceccato[1], Yoram Ofek[2], and Paolo Tonella[1]

[1] Fondazione Bruno Kessler—IRST, Trento, Italy
{ceccato, tonella}@fbk.eu
[2] University of Trento, Italy
ofek@dit.unitn.it

Abstract. The problem of software integrity is traditionally addressed as the static verification of the code before the execution, often by checking the code signature. However, there are no well-defined solutions to the run-time verification of code integrity when the code is executed remotely, which is refer to as run-time remote entrusting. In this paper we present the research challenges involved in run-time remote entrusting and how we intend solve this problem. Specifically, we address the problem of ensuring that a given piece of code executes on an remote untrusted machine and that its functionalities have not been tampered with both before execution and during run-time.

1 Introduction

When the software industry discusses software integrity, the main focus is on the protection of static software modules (e.g., by verifying the signature of their originator). On the other hand, dynamic software authentication in real-time during execution is a known problem without a satisfactory solution. Specifically, how to ensure that trusted code base (i.e., the software as was specified and coded) is running on an untrusted machine at all times and that the original code functionality was not modified prior to or during execution, is an open research challenge. This issue of entrusting software components is crucial since software, computers and networks are invading all aspects of modern life.

The issue of executing software in a trusted computing (TC) environment has gained a great deal of attention recently, in particular, the TCG (Trusted Computing Group) [22], Microsoft NGSCB (Next Generation Secure Computing Base) [23] and TrustZone developed by ARM [24] (see the next related work subsection for more details). These activities are somewhat complementary and orthogonal to the work presented in this paper. The previous approaches are hardware-based, and consequently, will not be available on all existing machines. Our research hypothesis is that a solution can be designed at any layer as a software component enhancing the layer itself in a cost-effective fashion; in contrast, TC is invasive, since it requires special hardware on the "motherboard". The proposed novel paradigm for remote entrusting of software will be available as a general, platform-independent solution (i.e., it is non-monopolistic, thus more competitive). The solution adds another line of defense to complement the current hardware solutions; while Trusted Computing (TC) can help manage keys and verify the system integrity during startup, it offers little protection against an attacker that already has access to the machine.

The key research question in remote entrusting is: "How can the execution of a software component be continuously entrusted by a remote machine, albeit the software component is running inside an untrusted environment?" (This is refer to as the "remote entrusting problem").

The solution to the above research problem should be able to employ external hardware, such as, smart cards, but not as a mandatory component. Furthermore, this work investigates a novel methodology for solving this problem by employing a software-based trusted logic component on a remote untrusted machine that in turn authenticates its operation continuously during run-time (i.e., execution). The method should assure the entrusting component that if the authentication is successful, then the original (i.e., unchanged) software functionality is being executed.

The long-term objective of the proposed approach is to entrust selected functionalities that are executed on untrusted machines and thereby ensure crucial trust/security properties.

Examples of possible applications are:

1. Protecting network resources and servers from users employing untrusted (i.e., unauthorized) software and protocols — specifically in the critical applications, such as, e-commerce, e-government.
2. Ensuring data privacy in Grid computing as well as digital right management (DRM) adherence by assuring proper processing of untrusted (possibly misbehaving) machines.

There are two fundamental differences between remote entrusting and other related approaches. Those fundamental differences are clear manifestation of some of the advancements beyond the state-of-the-art proposed by our approach.

1. *Core of trust location* — the basic working assumption when dealing with trust is that "some system components can be trusted" called at times, "core of trust". In some current approaches, such as trusted computing (TC), the "core of trust" is located locally on the "mother board", while in RE-TRUST the "core of trust" is placed in a remote trusted entity across the network. In other words, our model intend to address the trust problem by using the network under the assumption of continuous network connectivity, which is almost a reality today.
2. *Entrusting/validation method* — our proposed validation method is a significant departure from the-state-of the-art by introducing a novel protocol that provides software trust (or authentication) that is continuous during run-time — in other words, we introduce a proactive (avoidance) method. The main current approach to trust, e.g., TC (trusted computing) is off-line or reactive (after the fact); namely, it may be possible to detect trust violations after some damage has been done. The objective of RE-TRUST project is to avoid breach of SW trust damages all together.

In the rests of the paper describes how we intend to address the remote entrusting problem. In Section 2 the basic remote entrusting approach is described and in Section 3 more details are given regarding the general architecture, pure-software and the hardware assisted solutions, then Section 4 introduces some possible attacks and discusses some possible protection mechanisms. The discussion in Section 5 concludes the paper.

1.1 Related works

The initial work concerning the remote entrusting was developed by some consortium members in the TrustedFlow research activities [1,2]. The initial work introduced the ideas on how to generate a continuous stream of signatures using software only. The remote entrusting methodology is novel and challenging, and presents a major advancement beyond the state of the art. However, some specific aspects of the proposed research activities have been dealt with in different contexts. Therefore, the following state of the art discussion is divided into several subsections corresponding to various research aspects that are related to our approach.

Software dependability state of the art Software dependability is a mature and well-established research area that seeks solutions to the problem of software errors that can corrupt the integrity of an application. To this aim, several techniques have been developed and the most prominent are control-flow checking and data duplication. Control-flow checking techniques are meant to supplement the original program code with additional controls verifying that the application is transitioning through expected valid "traces" [3,4,5]. In data duplication techniques, program variables are paired with a backup copy [6,7,8]. Write operations in the program are instrumented to update both copies. During each read access, the two copies are compared for consistency. There is one main difference regarding the "attack model", between software dependability and the current project. Software dependability assumes that modifications are accidental (random) errors (say bit flips), while remote entrusting deals with intentional and malicious software modifications.

Software tamper resistance state of the art Among the several possible attacks, the focus is on the problem of authenticity, i.e., attacks aiming at tampering with application code/data for malicious purposes, like bypassing licensing, or forcing a modified (thus unauthorized) execution. Different solutions have been proposed in the literature to protect software from the above-mentioned rogue behaviors. Such solutions are surveyed in details in [9,10] and briefly described in the following. Obfuscation is used to make application code obscure so that it is complex to understand by a potential attacker who wants to reverse engineer the application. Obfuscation techniques, change source code structure without changing its functional behavior through different kinds of code transformations [11,12]. Theoretical studies about complexity of reverse engineering and de-obfuscation are in early stage. It is well-known that for binaries that mix code and data disassembly and de-compilation are undecidable in the worst case [13]. On the other hand, some work reported that de-obfuscation (under specific and restrictive conditions) is an NP-easy problem [14]. Further, it was proven that a large number of functions cannot be obfuscated [15].

Replacement background state of the art Dynamic replacement strategy relies on the assumption that tampering attempts can be made more complex if the attackers have to face newer versions continuously. This approach has some similarities with software aging [16], where new updates of a program are frequently distributed. This limits the spread of software "cracks" and it allows renewal of software protection techniques embedded in the application. Another relevant area of related work is represented by techniques for protection of mobile agents [16,17]. For instance, previous work proposed a scheme to protect mobile code using a ring-homomorphic encryption scheme based on CEF (computation with encrypted functions) with a non-interactive protocol [18,19]. However the existence of such homomorphic encryption function (also known as a privacy homomorphism) is still an open problem. Furthermore, some approaches mix obfuscation and mobility. For instance, in [20] agents are periodically re-obfuscated to ensure that the receiving host cannot access the agent state.

Hardware-based entrusting state of the art Solution proposed by Trusted Computing initiatives [21,22,23,24] rely both on a trusted hardware component on the motherboard (co-processor) and on a common architecture that enable a trusted server-side management application to attest the integrity of a machine and to establishing its "level of trust". This non run-time approach has been applied to assess integrity of a remote machine enhanced with a trusted coprocessor and a modified Linux kernel [25]. In that work a chain of trust is created. First BIOS and coprocessor measure integrity of the operating system at start-up, then the operating system measure integrity of applications, and so on. Other non run-time approaches rely on additional hardware to allow a remote authority to verify software and hardware originality of a system [26]. Beside Trusted Computing, another interesting approach is presented in [27]. This approach has some similarities to our hardware assisted method, as it is based on commodity hardware tokens (e.g., smart cards) and remote execution of selected software components.

2 Basic approach

Detection of software changes on a 1st machine by a 2nd machine across the network is difficult since the 2nd machine cannot directly observe the software executed on the 1st machine. As shown in Figure 1, in order to solve the problem, the 2nd machine should receive some "proofs" regarding the authenticity of the software that is running on the 1st machine. Such "proofs" are hard to obtain since the 2nd machine often receives only data from the 1st machine, while what is actually needed by the 2nd machine is to receive signatures (or attestations) continuously from selected parts of the software running on the 1st machine, i.e., selected applications and protocols that are executed on the 1st machine. The signatures (or attestations) will thereby authenticate the respective selected software parts executed on the 1st machine. In other words, the signatures that are continuously emanated from selected parts of the software on the 1st machine provide the "identity" of the running software and thereby enabling the 2nd machine, after validation, to entrust the software running on the 1st machine. However, today selected applications and protocols that are developed and deployed on such 1st machines are not designed to emanate signatures (or attestations) continuously. In essence this is a paradigm shift and one of the main scientific/technical challenges introduced in the RE-TRUST project [29].

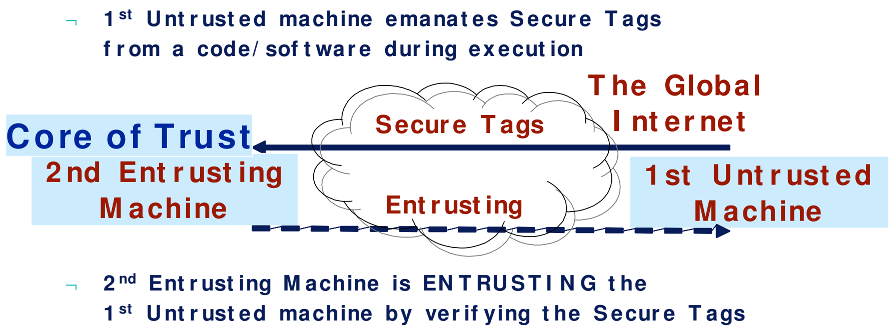
1st Untrusted machine emanates Secure Tags from a code/software during execution
The Global Internet
Secure Tags
Core of Trust
2nd Entrusting Machine
1st Untrusted Machine
Entrusting
2nd Entrusting Machine is ENTRUSTING the 1st Untrusted machine by verifying the Secure Tags

Fig. 1. Entrusting by remote software authentication during execution.

As stated before, networking and computing are converging into one system, consequently, various security and trust problems are emerging. The core of the remote entrusting principle (or entrusting, for short), presented in this research project, is: "*To utilize trusted entities in the system/network (firewall, interface, server, protocol client, etc.) in order to entrust selected software components in otherwise untrusted machines across the network, assuring their on-line/run-time functionality*". Namely, entrusting is based on the assumption that there are trusted entities in the converged system of networking and computing (obviously, if nothing can be trusted, building any trust relationship is not feasible). The term "untrusted machine" implies that a malicious user has access to system resources (e.g., memory, disks, etc.) and tools (e.g., debuggers, disassemblers, etc.) on the 1st untrusted machine, and consequently, is capable of tampering/modifying the authentic (i.e., original) code prior to or during execution. In other words, the objective is that a 2nd entrusting machine (e.g., "Core of Trust", see Figure 1) will entrust the 1st untrusted machine by "authenticating its execution" (i.e., in real-time). Indeed, the execution of software (code/protocol) is authentic/trusted if and only if its functionality has not been altered/tampered by an untrusted/unauthorized entity, both prior to execution and, more importantly, during run-time. Finally, note that the basic remote entrusting scheme depicted in Figure 1, can be extended to contemplate:

– Mutual remote entrusting: where the 1st and 2nd machines are entrusting one another.
– Transitive remote entrusting: where a 1st machine is entrusting a 2nd machine and a 2nd machine is entrusting a 3rd machine.

3 General architecture

The scientific and technical challenges involved in the present approach follows three orthogonal dimensions represented in Figure 2. Two dimensions represent two main software only approaches(code tamper resistance and code replacement), while the third represents the hardware assisted approach (tamper resistance (TR) using combined hardware (e.g., smart cards) and software).

The first software based dimension is the tamper resistance quality, it measures how difficult is to apply malicious modifications to the running program. Several techniques could be applied to increase the protection along this dimension, such as obfuscation, to increase the reverse engineering effort required to apply any attack.

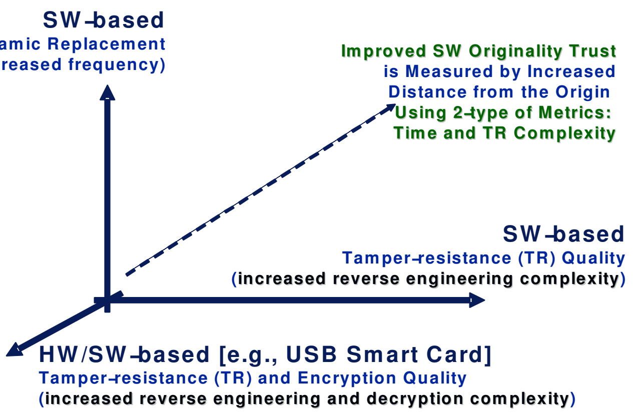
SW-based
Dynamic Replacement
(increased frequency)
Improved SW Originality Trust
is Measured by Increased
Distance from the Origin
Using 2-type of Metrics:
Time and TR Complexity
SW-based
Tamper-resistance (TR) Quality
(increased reverse engineering complexity)
HW/SW-based [e.g., USB Smart Card]
Tamper-resistance (TR) and Encryption Quality
(increased reverse engineering and decryption complexity)

Fig. 2. Quality of remote entrusting.

The second software pure-software dimension involves dynamic replacement. A portion of the application is periodically replaced at runtime, in order to give an attacker a limited time to complete an attack. Ideally a solution for remote entrusting should take advantage of both these two dimensions, because the more resilient is a tamper proof technique, the longer an attack would take to break it, the lower replacement frequency is required.

In the third dimension, tamper resistant methods involve co-design of application with software and hardware components, analyzing trade off between hardware and software. Pure software-based techniques may be extended to take advantage of the hardware dimension to increase the level of protection.

3.1 Pure software approach

The pure software dimensions investigate software-only methodologies for realizing the above-mentioned principle. In particular two objectives are addressed: (1) the secure software monitor should be combined (interlocked) in a secure way with the original application, and (2) the combined monitor must be robust against tampering (i.e., tamper resistance - TR). The first challenge will be dealt with by means of SW dependability techniques (e.g., for software faults detection). Tampering attacks are similar to random faults with the major difference that they are intentional (not accidental). Consequently, software dependability techniques are applicable to the trust domain as defined in Section 2. Finally, note that software dependability techniques are traditionally applied to a compiled code (e.g., C and C++). An additional challenge in this task will be to extend the above-mentioned techniques to an interpreted code (e.g., C# and Java). The second above objective will be addressed with two complimentary techniques: tamper resistance through software-based techniques, like source and binary obfuscation, and tamper avoidance, by dynamically replacing (parts of) secure software monitor, hence limiting the monitor lifetime (thus, also the tampering duration).

3.2 Hardware assisted approach

The hardware assisted dimension investigates tamper resistance methodologies combining hardware and software. With this approach, relatively inexpensive and widely available hardware monitors, such as smart cards or Trusted Platform Modules (TPMs) can be used to strengthen and improve the software-only protection method. A wide spectrum of possible solutions will be investigated ranging from low to high trust protection. This ranges from the hardware performing only some central operations (e.g., public key cryptography) to directly controlling the execution of major parts of the application, where the (untrusted) computer only stores encrypted code and data.

To investigate the combination of hardware- and software based software protection. The idea is twofold. On one hand, it is desired to utilize cheap and available hardware that alone may not be able to provide enough functionality. For example, trusted platform monitors, to strengthen the software protection. At the other extreme, the hardware itself may control most of the program flow, delivering maximum security at the price of a performance penalty. Finally, it is desired to investigate solutions in between the two extremes, to allow developers to freely choose the trade off between hardware cost, performance and security. Along the above lines, two major issues need to be investigated:

1. Regarding the low trust protection mode, execution of the code must be split between hard- and software, in a way that maximizes protection and minimizes the performance penalty.
2. Regarding the full trust protection, methods must be developed that allow an attacker to observe the entire communication between the computing engine (the secure hardware) and the memory (in the PC), without learning any useful information.

Novel methods should be developed to scale the protection level, i.e., to discreetly adapt the trust and security level of specific scenarios.

3.3 Monitor

The secure software monitor and the original application must be correlated in such a way that any attempt to corrupt the authenticity of the application will be detected by the monitor, and that any attempt to harm the integrity of the monitor itself will stop the generation of valid signatures. This constitutes a major and open challenge in a pure software methodology. The initial approach will investigate techniques borrowed from the software dependability discipline, as in the area of software fault tolerance.

Innovative methods will be investigated to exploit the "time dimension" to increase overall tamper resistance of the secure software monitor. Namely, to bound the time available for attackers by means of dynamic software updates, where (parts of) the secure software monitor can be replaced at any instant during run-time. This approach improves tamper avoidance by making the life-time of each secure software monitor to a short defined time interval. To achieve this goal two major issues must be investigated, i.e., replacement strategies for interpreted (e.g., C# and Java) and compiled (e.g., C and C++) code the automated and non-predictable generation of secure software.

In pure software methods, the secure software monitor has to be protected from tampering. In particular, this requires solutions to two different problems: (1) the monitor behavior must be hidden to avoid trivial reverse engineering, and (2) secret data inside the monitor (e.g., encryption keys) must be hidden in order to be not easily spotted. It is envisioned that the obfuscation and white-box cryptography are the means to address the above-mentioned problems, respectively. This problem is articulated in the following ones: source-to-source obfuscation, obfuscation of (Java) byte code and protection of embedded keys with white-box cryptography techniques.

4 Attacks and analysis

This sections introduces some possible attacks and then discusses some possible protection mechanisms.

4.1 Possible attacks

A number of attacks may be applied to the remote entrusting software/hardware protection schemes. The attacker objective is to prevent the trusted machine from detecting tampering, and consequently, (remote) entrusting an untrusted machine. The rest of the section uses P to denote the program running on the untrusted machine that must be protected. The secure monitor will be called M.

Reverse engineering attacks Reverse engineering attacks aim at locating important functionalities and data both in P and M. Once located, functionalities and data are altered maliciously. Key functionalities and data that can be the target of a reverse engineering attack in the remote entrusting scheme are:

- Secure tag (sequence) generator.
- Authenticity checking functions.
- Secret keys (e.g., used for secure tag generation).
- Input data (e.g., passed to checking functions).
- Output data (e.g., produced by checking functions).

The attacker may attempt to locate the function of M that generates the authenticity secure tag sequence and any secret key used for it. Once located, this functionality could be tampered with in order to produce correct authenticity tags even when P should not be trusted.

The attacker may attempt to locate and tamper with the functions that are devoted to checking the authenticity of P and of the underlying software and hardware. They could be modified so as to return a positive test result even when the actual result is negative. Moreover, they could be analyzed to understand which parts of the whole system are subjected to frequent checks and which ones are verified more rarely, in order to discover weak points where to apply a malicious modification that would be revealed with low probability.

The attacker may attempt to locate and change the input and output values involved in function calls, so as to change the behavior of the called function. This could be used to alter the result of a check performed by the monitor M or to tamper with the expected behavior of the program P. One way to change inputs or outputs is by directly modifying the code. Another possibility, which requires a combined execution environment attack (see next section), consists of intercepting the function call and changing input or output values dynamically. This second approach is stealthier than the former, because it cannot be detected by a code analysis on P. In fact, the running code is the original one.

Execution environment attacks The attacker can tamper with the underlying execution environment, thus altering the behavior of P without modifying its code. Instead of deploying P on the actual processor, the attacker could run it on a simulated processor, which implements the same functionalities of the actual processor in software. It provides registers, interrupts and I/O devices. It can interpret and execute binary code. The simulated processor can be stopped when specific events occur. The current context (*i.e.*, memory, call stack, parameters) can be analyzed and modified. Then, execution is resumed. The attacker can take advantage of this infrastructure to intercept calls to libraries, to operating system and I/O facilities in order to dynamically modify parameters and memory locations and, thus, maliciously change the behavior of the program.

A similar attack consists of executing program P inside a debugger, which traces all the executed instructions, memory accesses and memory content. The debugger can interrupt the execution when selected instructions or conditions are met and the user can perform dynamic modifications.

Another attack to the execution environment may be directed toward the dynamic libraries. By altering them, it could be possible to trace and modify any operation that the program delegates to them, such as I/O, memory management, file system storage and network communication.

Cloning attack In a cloning attack two copies of P are installed. The first copy is the original, unadulterated one. The monitor M is correctly installed with it and it is periodically updated. The execution environment, operating system and hardware are genuine. Such a program sends the server the expected authenticity secure tag sequence and, thus, is entrusted. The attacker maliciously modifies the second copy of P. The tampered copy runs in parallel with the first one, but not necessarily on the same client. The output of its monitor M is simply discarded.

All the network traffic coming from the server is sent both to the original and to the tampered applications (*e.g.* using a modified network device), so they can be executed in parallel on the same input values. The original application provides the authenticity tags required to be trusted by the server, whereas the tampered copy provides the modified behavior required by the malicious user.

The effectiveness of this attack depends on the possibility to decouple the generation of the tag sequence from the communication occurring between client and server. In fact, if the secure tag sequence originated from P includes data used by the server to carry out the computation required by the client (as prescribed by some variants of the remote entrusting scheme), a consistent computation must be performed on original and tampered copy, so as to keep unchanged the communication with the server. There are several classes of network applications for which the preservation of the communication between client and server entails that no malicious tampering is actually taking place (e.g., no unfair behavior can take place, no incorrect billing can be originated, etc.). One important class of applications for which preservation of the communication is not enough to ensure that no malicious tampering is taking place is Digital Right Management (DRM). In fact, DRM applications should prevent the client from creating illegal copies. However, creating an illegal copy does not involve any communication between client and server.

Differential analysis attack Differential analysis consists of gathering information about the monitors by comparing the sequence of monitors produced and delivered by the monitor factory over time. Previously released monitors may be successfully broken when new ones are delivered, since the attacker has more time to reverse engineer them. If their analysis reveals, to some extent, the strategy implemented by the monitor factory, the attacker could take advantage of this knowledge to reduce the time necessary to break the current monitor, eventually compromising it before its expiration time.

Dependencies among attacks The attacks described above are not independent of each other. In particular, the execution environment, cloning and differential analysis attacks all depend on the reverse engineering attack and cannot be performed without it.

With regards to the execution environment attack, deciding when to intercept the execution and how to alter it depends on the goal of the attacker. In turn, this requires some level of understanding of the program being tampered with. Potentially, a huge amount of information can be gathered and modified at run time. Focusing on the relevant functions, data and events requires a deep level of knowledge about the running code, hence the dominant problem becomes reverse engineering. Thus, modification of the execution environment is just a way to implement the reverse engineering attack.

A similar argument holds for cloning and differential analysis attacks. In the cloning attack, substantial reverse engineering effort must be devoted to locating the functions to be tampered in the program and to ensure the correct authentication secure tag sequence is sent to the server. Comparison between successive versions of the monitor aims at simplifying its reverse engineering, which is at the core of the possibility to modify it maliciously.

Overall, the attack model consists of a specific technique to gather and alter information (either adulteration of execution environment, cloning or differential analysis) combined with the understanding of the program and the monitor, to be achieved through reverse engineering.

4.2 Analysis of attack resistance

The trust model, and its variants, currently defined in the remote entrusting scheme address in various ways the attacks described in the previous section. In this section each source of trust in the trust model is related to the attacks it provides some defense against. The strength of such a defense is also briefly discussed.

Table 1 relates sources of trust to attacks. The meaning of a cell marked `X` is that the corresponding source of trust contributes to some extent to defend against the attack in this cell column. This does not mean that the source of trust provides full or proved protection against an attack. The given defense makes the attack harder, by addressing the vulnerabilities exploited by the attack, but it may still be only a partial defense.

For example, code obfuscation provides a limited defense against reverse engineering. In fact, an obfuscated code is expected to be harder to understand

and analyze than a clear text. However, given enough time, a determined attacker can always reverse engineer a program, despite its code obscurity. Combined with monitor replacement, code obfuscation becomes a stronger defense, since only a limited time is available to the attacker to de-obfuscate the code.

Verification of the integrity of P (first row of Table 1) is a protection against malicious modifications that do not involve the monitor M. It must be combined with the verification of the verifier (M) itself to become effective against attacks that involve modifications of the monitor (columns 2, 3, 4, 6). Verification of the libraries (row 3) is an important defense line against execution environment attacks. However, more checks are needed to verify that HW/OS are genuine and that execution is not in debug mode. Direct implementation of this defense might be problematic and hard to achieve, since the monitor has limited capabilities of testing the HW/OS and the execution mode when running on the client. However, further lines of defense can be put in place. Firstly, the verification of the output of computations that depend on HW/OS and execution mode can be exploited for this purpose (see row 6). Moreover, since modification of the execution environment must be necessarily combined with a reverse engineering attack, reverse engineering resistance mechanisms, such as code obfuscation, provide a defense against execution environment attacks as well, in an indirect way.

The verification of the results of selected computations is also effective against modifications of P, M, as well as modifications of data passed to or returned from genuine functions of P and M. Hiding a secret key into the monitor is essential to protect the secure tag sequence generator (row 7). Monitor replacement (row 8) is an important countermeasure against tampering with the monitor (columns 2-7). Combined with the capability of producing new monitors that are independent from the previous ones, monitor replacement gives also some protection against the differential analysis attack.

Reverse engineering resistance, of which code obfuscation is one possible instance, is potentially effective against any attack, either directly or indirectly, because any attack must be necessarily combined with reverse engineering and program understanding, in order to alter the program behavior in a meaningful way, according to the attacker's goals.

The network of trust (row 10) increases the protection of the monitor's code (columns 2, 3, 4, 6). Inclusion of output data into the secure tags makes replacement of the checking function harder, as well as tampering with the output data produced by functions. Making the communication with the server bi-directional makes any change to the monitor harder (columns 2-7), since such changes might affect the verification triggered by the challenge, invalidating the result. If the verification is coupled with the ongoing computation, because the secure tags include a portion of the output (row 11), the cloning attack can be effective only if the communication with the server is preserved in the clone, which means that for many classes of applications no malicious tampering can actually take place at all.

5 Discussion

In this paper we presented the problem of remote entrusting as a novel paradigm, which give rise to multiple interdisciplinary problems encompassing many aspects of computing and networking. The central issue is how to entrust a piece of software executing on an untrusted and remote machine. We proposed a general architectural framework for remote entrusting with three problems: (*i*) how to combine two programs (the original code with the secure tags generation code) into one combined program, (*ii*) how to make it hard separate using reverse engineering techniques the combined program, and (*iii*) how to dynamically replace parts of the combined program during run-time in order to limit the time available for an attacker to reverse engineer the combined program. The previous three problems are investigated and solved along three main research dimensions, which require comprehensive solution. Specifically, dynamic replacement, tamper resistance with and without hardware assistance. A number of attacks have been identified on the proposed architectural framework, they have been analyzed and discussed.

Effective solutions of the remote entrusting problem will impact many application areas. Two main categories of applications have been identified,depending on the direction of the flow of data. The first category contains all the applications where the untrusted client sends data to the trusted machine (e.g., server) and the latter reacts by delivering a certain service (e.g., e-commerce, e-government). For these applications, a viable solution is based in hiding secure or authenticity tags in the outgoing data, in such a way that it would be difficult to tamper with them without affecting the data. Applications in the second category are those ones that receive private or protected data from the trusted party (e.g., grid computing, digital right management). In this case the solution is more challenging because, once delivered, protected data can not be longer protected, even if a tampering is detected.

Acknowledgments. This work was supported by funds from the European Commission (contract N^o 021186-2 for the RE-TRUST project, see [29])

References

1. M. Baldi, Y. Ofek, M. Young: Idiosyncratic Signatures for Authenticated Execution of Management Code. 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 2003), Heidelberg, Germany, Oct. 2003.
2. M. Baldi, Y. Ofek, M. Young: The TrustedFlow(TM) Protocol - Idiosyncratic Signatures for Authenticated Execution. 4th Annual IEEE Information Assurance Workshop, West Point, NY, USA, June 2003.
3. N. Oh, P.P. Shirvani, E.J. McCluskey: Control-flow checking by software signatures. IEEE Transactions on Reliability, Vol. 51(1), Mar. 2002
4. J. Ohlsson, M. Rimen: Implicit signature checking. Proceedings of 25th International Symposium on Fault-Tolerant Computing, Jun. 1995

5. A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, L. Tagliaferri: Control-flow checking via regular expressions. Proceedings of 10th Asian Test Symposium, Nov. 2001
6. N. Oh, S. Mitra, E.J. McCluskey: ED4 I: error detection by diverse data and duplicated instructions. IEEE Transactions on Computers, Vol. 51(2), Feb. 2002
7. N. Oh, P.P. Shirvani, E.J. McCluskey: Error detection by duplicated instructions in super-scalar processors. IEEE Transactions on Reliability Vol. 51(1), Mar. 2002
8. A. Benso, S. Chiusano, P. Prinetto, L. Tagliaferri: A C/C++ source-to-source compiler for dependable applications. Proceedings of International Conference on Dependable Systems and Networks (DSN), Jun. 2000.
9. C. Collberg, C. Thomborson, D. Low, Watermarking: Tamper-Proofing, and Obfuscation - Tools for Software Protection. IEEE Transactions on Software Engineering, vol. 28, 2002
10. G, Naumovich, N. Memon: Preventing piracy, reverse engineering, and tampering. IEEE Computer, vol. 36(7), pp. 64 ?71, July 2003
11. C. Wang, J. Davidson, J. Hill, and J. Knight: Protection of software-based survivability mechanisms. Proceeding of International Conference on Dependable Systems and Networks (DSN), Goteborg, Sweden, July 2001
12. E. Valdez and M. Yung, Software DisEngineering: Program Hiding Architecture and Experiments. Information Hiding, 1999
13. C. Linn, S. Debray: Obfuscation of Executable Code to Improve Resistance to Static Disassembly. Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS), Oct. 2003
14. A. W. Appel: Deobfuscation is in NP. www.cs.princeton.edu/ appel/papers/deobfus.pdf
15. B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, K. Yang: On the (Im)possibility of Obfuscating Programs. Proceedings of CRYPTO, 2001
16. G. McGraw, E.W. Felten: Mobile Code and Security. IEEE Internet computing, 1998 (Vol. 2, No. 6)
17. Oscar Esparza, Miguel Soriano, Jose L. Munoz, Jordi Forne: Detecting and Proving Manipulation Attacks in Mobile Agent Systems. Lecture Notes in Computer Science, Volume 3284, Jan 2004, Pages 224-233
18. T. Sander and Christian F. Tschudin: Towards Mobile Cryptography. IEEE Symposium on Security and Privacy, May 1998
19. T. Sander, C. F. Tschudin: Protecting mobile agents against malicious hosts. Lecture Notes in Computer Science, 1998
20. L. Badger et al.: Self-protecting mobile agents obfuscation techniques evaluation report. NAI Labs Report, Nov. 2001, online at www.isso.sparta.com/research/documents/spma.pdf
21. S. Pearson: Trusted computing platforms, the next security solution. Technical Report HPL-2002-221, HP Laboratories, 2002
22. The Trusted Computing Group: On-line at https://www.trustedcomputinggroup.org
23. Next Generation Secure Computing Base, http://www.microsoft.com/resources/ngscb
24. R. York: A New Foundation for CPU Systems Security. ARM Limited, http://www.arm.com
25. R. Sailer, X. Zhang, T. Jaeger, L. van Doorn: Design and Implementation of a TCG-based Integrity Measurement Architecture. Proceedings of the 13th USENIX Security Symposium San Diego, CA, USA, Aug. 2004
26. Rick Kennell, Leah H. Jamieson: Establishing the Genuinity of Remote Computer Systems. Proceedings of the 12th USENIX Security Symposium, 2003

27. A. Mana, J.Lopez, J. Ortega, E. Pimentel, J.M. Troya: A Framework for Secure Execution of Software. International Journal of Information Security, Vol. 3(2), 2004
28. A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. K. Khosla: Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP), Brighton, UK, October 23-2-6, pages 1–16, 2005.
29. http://re-trust.org/

Sources of trust	Attacks											
	Reverse engineering attacks							Execution environment attacks			Cloning attack	Differential analysis
	P is tampered with	Replace checking function	Replace tag sequence generator	Modify input before call on		Modify output before return on		Replace HW/OS	Replace dynamic libraries	Tampered execution (debug mode)		
				M/P	env.	M/P	env.					
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)
(1) M checks P text and data segment	X											
(2) M self checks itself before checking P		X	X	X		X						
(3) M checks libraries used by P									X			
(4) M checks execution environment					X		X			X		
(5) M checks the OS and the HW								X				
(6) M checks results of computation	X			X	X	X	X	X	X	X		
(7) Secret key used to generate the tag sequence			X									
(8) Monitor replacement		X	X	X	X	X	X					X
(9) Rev-eng resistance (code obfuscation)	X	X	X	X	X	X	X	X	X	X	X	X
(10) Network of trust (self-checking implementation)		X	X	X		X						
(11) Tags include (portion of) output		X				X	X				X	
(12) Bi-directional communication (challenge from the server)		X	X	X	X	X	X					

Table 1. Sources of trust related to specific attacks