

Entrusting Remote Software Executed in an Untrusted Computation Helper

Moez Ben MBarka^{*†}, Francine Krief[†] and Olivier Ly[†]

^{*}Cryptolog

6 rue basfroi, 75011 Paris - France

moez.benmbarka@cryptolog.com

[†]LaBRI, University of Bordeaux 1

351, cours de la Libération - 33405 Talence - France

{krief, ly}@labri.fr

Abstract—How to trust an application executed in a remote untrusted client? Indeed, in an untrusted environment, an attacker may tamper with the application code or the execution environment to alter the application behavior for its own purposes. This problem is traditionally addressed by checking the integrity of the application code at loading and during runtime. However, this line of protection is not sufficient when the client is used as a computation helper and is expected to return the result to a trusted server. An attacker may execute the application without altering its code but returns an invalid result. This paper proposes a new approach to deal with remote software execution in an untrusted client used as a computation helper. The proposed solution provides the integrity of the computation result returned to a secure server.

I. INTRODUCTION

Remote trust computing is a very difficult problem which has gained wide interest recently. The main issue is simple and boils down to: how to trust a code executed in a remote untrusted environment. By “untrusted environment” we mean a computing environment in which a malicious user has a complete access to the system resources (memory, io devices...) and tools (debuggers, de-obfuscation tools...) to try to reverse-engineer or alter the code for its own purposes.

With the growing popularity of network based computations, remote trust would have many applications. For instance, entrusting remote computation is an important requirement for grid computing and mobile agents [1], [2]. Ideally we would like to provide black-box security [3]: it means that a malicious user controlling the untrusted machine can neither understand the computation function or data (input/output) nor alter them. This security level may be achieved using computing with encrypted functions or computing with encrypted data schemes [4], [5]. The main advantage of these techniques is that they have theoretically provable strength. However, the current results are limited to the evaluation of polynomial and rational functions and therefore can not be used in most real applications.

Another approach is to try to make the task of a potential attacker as hard as possible using heuristic techniques such

as software obfuscation [6]–[8] and white-box cryptography techniques [9]. While the security level of such techniques can not be proven formally, they have the advantage to be more general and applicable to any computation. This paper will address the latter approach: we do not try to achieve a formal level of security, but rather a practical level of protection acceptable for many applications.

The rest of the paper is organized as follows. Section II describes a simple computation model for which remote trust is required. Section III presents a novel approach to deal with the problem of remote software trust described in recent papers [10], [11]. Section IV describes the contribution of this paper to improve this approach. Section V concludes the paper.

II. REMOTE COMPUTATION MODEL

We will consider the following scenario: Bob wants to compute a function F over an input x but he does not have the sufficient computation power in its server S . He writes $F(x)$ as $G(H(x))$ where G can be efficiently computed by S and asks Alice to compute H in its powerful machine C (will be denoted as a client). Then, for each input, the computation is as follows:

- S sends x to C
- C computes $y = H(x)$ and sends it to S
- S computes the final result $z = G(y)$

C is considered untrusted and under the control of a potential attacker. In the scope of this paper, we will assume that the remote computation can be trusted provided the received result is correct. We are not interested in code or data confidentiality.

As stated before, we do not look for an absolute security level but for acceptable protection solutions given the constraint that the client is considered untrusted and possibly hostile. Moreover, we look for a software-only based solution which can be deployed on any helper machine with no need to specific hardware or operating system capabilities.

III. STATE OF THE ART

Many solutions are proposed in the literature based on software protection mechanisms [3], [6], [7], [12].

The first line of protection is to check the integrity of the code at load time. This approach is obviously not sufficient since an attacker may still alter the computation at runtime. An improvement is to check the integrity of the code continuously at runtime to detect any modification after code loading.

However, most of these solutions are off-line [3], [6]. It means that they check the integrity of the code but not the integrity of the result returned to S . For instance, Alice may execute the code without altering it but intercepts its result and sends an incorrect one to S .

To also have the result integrity, the client output must be bound to the code which returned it in a trusted way. Therefore, in addition to remote software integrity, we need to have remote software authentication. In other words, the server must be able to authenticate the code being executed in the remote client and to verify its integrity at runtime. Obviously, this is not an easy problem, since the code is executed in a remote client under the control of potentially malicious attacker.

This section will outline an approach described in [10], [11] to deal with remote software authentication and integrity. The contributions of this paper to improve this approach are described in the next section.

A. Remote software monitoring

Recent papers [10], [11] addressed this problem by combining protection techniques from many research areas. Code integrity is addressed by embedding a code checker called monitor [10] within the remote application. This monitor is assumed to be trusted and it checks the application integrity in addition to its own integrity at startup and during runtime. Moreover, the monitor can check the integrity of the execution environment (hardware, system calls, debugger tools, execution time...) to detect any tampering attempt.

Remote authentication is addressed by sharing a secret key between the server and the application. This key is hidden in the trusted monitor. White-box cryptography techniques [9], [13] are used to encrypt data without revealing the encryption key. The monitor should be inter-locked with the original application code using obfuscation techniques in a way that it is not easy to use the monitor to encrypt messages other than ones produced by the original application.

The trust is achieved through secure tags emanating from the monitor (see figure 1). Provided the monitor does not detect any tampering attack, it continuously generates encrypted secure tags. As valid tags are received, the server can assume that: the result was computed by the original code and that code was not altered. Therefore, the result can be trusted to be correct. Once the monitor detects any attack attempt (code modification, invalid environment execution parameter...) it generates invalid tags to make the server aware of the attack. Note that this approach is not appropriate for off-line applications (for instance, software intellectual property protection) where the client does not have any interaction with a remote trusted server and thus its conformity to the expected behavior can not be checked remotely.

However, it is clear that the above techniques make the task of an attacker harder but not impossible. Provided he has enough time, an attacker may break the code obfuscation or inter-locking or access to the hidden secret key. To make any tampering attack harder, dynamic code replacement techniques can be used. To do so, the server periodically updates the monitor code. The new monitor should be generated in a way that it can not be broken automatically by an attacker who has successfully broken the previous one.

To summarize, four protection techniques are combined to provide a practical security level for remote computing:

- 1) code obfuscation.
- 2) software runtime monitoring.
- 3) white-box cryptography.
- 4) dynamic code replacement.

The role of the three first techniques is to make the task of an attacker as hard as possible. The last technique aims to guarantee that an attacker does not have enough time to break the previous protections since the attacked code (or parts of it) is periodically replaced. This relies on the assumption that tampering attacks are harder to conduct if the attackers have to face continuously changing versions of the code [3], [6].

B. Tampering attacks

Many attacks can be applied to any of the above protection mechanisms. Most of these attacks are surveyed in [10]. Naturally, the main target of the attacks is the trusted monitor. The attacker objective is to prevent the monitor from detecting tampering so that it continues to emanate valid secure tags even when the computation is altered. It is exposed to many kinds of attacks mainly based on:

- reverse-engineering attacks.
- differential analysis attacks.

Reverse-engineering attacks [14] may target hidden data (for instance, the secret key) or functionalities (for instance, functionalities responsible to check the execution and to generate authentication tags). Once some useful information about the monitor is revealed, the attacker may either target the monitor itself (by modifying its code) or tamper with the execution environment to alter the behavior of the monitor. The last technique is more difficult to detect. For instance, the attacker may run the original application and monitor without any modification but dynamically tamper with the execution context (by changing function inputs, memory locations...) to compromise the behavior of the monitor.

Differential analysis attacks aim to break at least a monitor version (for instance by applying reverse-engineering techniques on early versions of the monitor) and then to build an automatic tool which breaks any new version by comparing the sequence of the deployed monitors. Therefore, code replacement needs a sort of a monitor factory able to produce an infinity set of monitors which are functionally equivalent but with new embedded secret key and different verification strategy.

Figure 1. Secure tags generation.

IV. NEW APPROACH

In the model described in [10], the monitor is mainly composed of four parts:

- 1) data gathering function: collects the information about the code and the execution environment to be checked.
- 2) verification function: based on the collected information, it decides whether to continue to trust the client.
- 3) encryption function: generates the secure tags provided the verification function did not detect any tampering attempt.
- 4) secret data (keys and metrics).

The security of these components relies on combination of software obfuscation techniques and white-box encryption [15].

In this section, we will propose two improvements:

- 1) Reduction of the size of the monitor by limiting the functions and data needed to be embedded and protected.
- 2) Proposition of an authentication alternative to the use of secret key encryption for secure tag generation.

First, we propose to review our model. We assume that the client has passed a one-time registration phase before being used as a computation helper. During the registration phase, the server registers all useful information needed to monitor the remote computation. This includes platform information (for instance hardware settings) and execution environment information (for instance, system calls, expected execution time...). Therefore, we no longer need to embed these data within the monitor.

We assume that during the registration phase, the client is trusted. It means that all information given to the server during this phase are related to trusted and unaltered components. For instance, the registration phase can be conducted through an audit procedure preceding the deployment of the computation system.

A. Moving the monitoring strategy to the server

The monitor is the most critical and vulnerable defense point which may be targeted by an attacker. Accessing the monitor strategy may lead to more efficient attacks. For instance, if the attacker has information about which system attributes are checked, he may discover weak points to tamper with the execution environment leading to altering the

monitor checking inputs.

We propose to move most parts of the verification code to the server itself. The role of the monitor is then limited to data gathering only. It continuously sends the attributes (for instance code hashes, execution environment properties) to check within the secure tags to the server which based on these information decides whether to continue to trust the client or to immediately stop communicating with it. This relies on the fact that the server has collected during the registration phase all needed attributes and that these data are trusted.

This approach limits the need to replace the monitor code or at least limits the parts of it needed to be continuously updated. However, monitor replacement may still be needed. Indeed, the secret key is still the drawback of the all system security and should be periodically changed.

B. Authentication using an obfuscated hash algorithm

In the approach described in [10] and [11], the mutual authentication is based on sharing a secret key between the trusted monitor and the server. White-box cryptography techniques are used to do encryption in a potential hostile environment without revealing the key [9]. We propose the use of a dynamic secret hash algorithm instead of the secret-key encryption. The hash algorithm should be obfuscated and periodically updated in the monitor.

The secure tags are computed as shown in figure 2.

The input X is sent without any encryption. This does not matter since confidentiality is not a requirement. The secure tag is calculated using the input X , the output Y and the hash of the set of the attributes that the monitor continuously checks. The monitor has also a counter i which is incremented for each request and is included in the tags to avoid the use of a valid secure tag twice. The secure tag is generated for each request and concatenated to the corresponding response. Then, the server computes the same security tag based on the registered attributes and checks that it is equal to the tag returned by the client. Note that if X , Y or any verified attribute has been altered, the checking function in the server will fail and the attack will be detected.

This approach relies on the fact that the hash algorithm is obfuscated and then kept secret. Code replacement is still

Figure 2. Secure tags generation.

needed to prevent an attacker to have enough time to break it. If the obfuscation is broken for at least a monitor version, the attacker may access the hash function. Therefore, the monitor factory should change this algorithm for each version. We propose the use of a set of dynamic hash algorithms [16]. This type of algorithm takes two inputs instead of one. The additional parameter is a security parameter which specifies the internal behavior of the algorithm and the size of the output. As with most dynamical systems with a large number of states, generic parameters may yield sufficiently complex behavior of the hash algorithm. This may harden the task of an attacker to reverse-engineer the monitor based on its behavior.

The use of obfuscation and the need to generate a secure tag for each request obviously add time overhead on the computation of H at the client side. This overhead must be minimized as much as possible. Otherwise, it would be more efficient to compute both H and G at the server. The computation overhead depends both on the used obfuscation transformation and the tag generation algorithm. To optimize the tag generation, polynomial hash algorithms may be used. The Tillich-Zemor algorithm described in [17] and improved in [18] is a good candidate. This algorithm takes as security parameter a polynomial p which can be updated by the factory for each monitor version. Computing this hash function involves multiplication, addition and division of polynomials of degrees bounded by that of the parameter p . These operations are quite efficient and their time complexity is bounded by a constant which depends only on the choice of p .

C. Analysis

The components of the monitor are reduced to the two following functions which should be obfuscated:

- 1) dynamic hash function.
- 2) data gathering function.

We believe that more we reduce the functionalities and data to be included in the monitor, more the task of an attacker is hard. Furthermore, since the monitor is reduced to two main functions, updating the monitor code becomes easier and relies on periodically changing the hash algorithm security parameter.

As stated before, the security of the system is heuristic. It is based on the robustness of the obfuscation algorithms and the choice of the appropriate life duration of a monitor version. We believe that the improvements proposed in this paper improve the security of the system against reverse-engineering and differential analysis attacks.

The robustness of the system against reverse-engineering attacks relies on the use of code obfuscation techniques. Obfuscation techniques combined with the use of a dynamic hash algorithm may be used to check the integrity of the computation without using an encryption secret key algorithm which may represent an additional point of failure for the security of the system. Moreover, a secure tag has to be generated for each request. Therefore, a complex monitor is not suitable for use in an untrusted computation helper where efficiency is a determinant parameter. Using a polynomial hash function limits the computation overhead added by the monitor.

Reverse-engineering attacks based on altering the inputs of the monitor are also made harder since the attacker does not have access to any information about how the application and the execution environment attributes are verified. The verification strategy is executed on the trusted server and is assumed to be kept secret. Furthermore, it can change

at runtime. The attributes to verify depend both on the application context (execution environment and platform) and the required security level. More the set of the checked attributes is large, more the probability to detect tampering attacks is high.

Regarding differential analysis attacks, the use of dynamic hash functions may yield less vulnerable monitor. Updating randomly the algorithm security parameter for each monitor version affects the monitor behavior and output size. Changing this parameter may yield sufficiently complex behavior of the hash algorithm which makes attacks based on reverse-engineering and differential analysis harder. Therefore, building an automatic tool to break the monitors becomes more difficult.

V. CONCLUSIONS

This paper presented a new approach to trust a code executed on an untrusted computation helper. This approach is an improvement for techniques addressed in many recent papers [10], [11]. These techniques provide remote entrusting of client-side application code by continuously checking its integrity and authenticity at run time. This is achieved through continuously emanating secure tags from a trusted entity, called monitor [10]. The monitor is embedded in the application code and is continuously updated.

The paper proposed two improvements. First, we proposed to remove the full verification strategy to the trusted server. The main role of the monitor is reduced to gather security attributes (about the application code and its execution environment and platform) and to compute the secure tags. The trusted values of these attributes have been registered at the server during a registration phase.

The paper also proposed to use a dynamic hash algorithm instead of a secret key encryption. The dynamic hash algorithm takes a security parameter which may be updated for each new monitor version. Changing this parameter may yield sufficiently complex behavior of the hash algorithm which makes attacks based on reverse-engineering and differential analysis harder. Furthermore, since the client is used as a computation helper in the model presented in this paper, the computation overhead added by the monitor should be optimized as much as possible. We suggested the use of polynomial hash functions similar to the algorithm presented in [18].

The security of the system is obviously heuristic and relies on the robustness of the obfuscation algorithm and on the dynamic replacement of the monitor at runtime. However, we believe that we can achieve a security level appropriate for many applications, for instance grid computing.

The registration phase has not been detailed in this paper. Future work includes defining the procedure of this phase (how to collect trusted attributes of an untrusted execution environment) and the set of the attributes that should be

collected and checked to provide a satisfactory level of trust. Future work needs also to investigate the set of dynamic hash functions that provide the best level of security with an acceptable computation overhead to keep the computation at the client side efficient.

ACKNOWLEDGEMENT

The authors would like to thank the anonymous reviewers for their valuable comments.

REFERENCES

- [1] J. Algesheimer, C. Cachin, J. Camenisch, and G. Karjoth, "Cryptographic security for mobile code," in *Proceedings of the 2001 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2001, p. 2.
- [2] J. R. D. Dyson, N. E. Griffiths, H. N. L. C. Keung, and S. A. Jarvis, "Trusting agents for grid computing," in *In Proceedings of the IEEE SMC 2004 International Conference on Systems, Man and Cybernetics*, 2004, pp. 3187–3192.
- [3] F. Hohl, *Time Limited Blackbox Security: Protecting Mobile Agents From Malicious Hosts*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1998, vol. 1419, ch. chapter 6, pp. 92–113.
- [4] T. Sander and C. Tschudin, "Towards mobile cryptography," in *Proceedings of the IEEE Symposium on Security and Privacy*. Oakland, CA, USA: IEEE Computer Society Press, 1998.
- [5] T. Sander and C. F. Tschudin, *Protecting Mobile Agents Against Malicious Hosts*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1998, vol. 1419, ch. chapter 4, pp. 44–60.
- [6] B. Matt, A. Reisse, T. V. Vleck, S. Schwab, and P. Leblanc, "Self-protecting mobile agents obfuscation report," Network Associates Laboratories, Report 03-015, Tech. Rep., 2003.
- [7] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, University of Auckland, New Zealand., Tech. Rep., July 1997.
- [8] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," *Lecture Notes in Computer Science*, vol. 2139, 2001.
- [9] S. Chow, P. Eisen, H. Johnson, and P. C. V. Oorschot, "White-box cryptography and an aes implementation," in *Proceedings of the Ninth Workshop on Selected Areas in Cryptography (SAC 2002)*. Springer-Verlag, 2002, pp. 250–270.
- [10] M. Ceccato, Y. Ofek, and P. Tonella, "Remote entrusting by run-time software authentication," in *SOFSEM*, ser. Lecture Notes in Computer Science, V. Geffert, J. Karhumäki, A. Bertoni, B. Preneel, P. Návrat, and M. Bieliková, Eds., vol. 4910. Springer, 2008, pp. 83–97.
- [11] P. Falcarin, R. Scandariato, M. Baldi, and Y. Ofek, "Integrity checking in remote computation," in *Proceedings of "AICA", Udine, Italy October 2005*, 2005.
- [12] J. M. Memon, A. Khan, A. Baig, and A. Shah, *A Study of Software Protection Techniques*. Springer Netherlands, 2007, ch. chapter 45, pp. 249–253.
- [13] B. Wyseur, W. Michiels, P. Gorissen, and B. Preneel, "Cryptanalysis of white-box des implementations with arbitrary external encodings," 2007.
- [14] M. Madou, B. Anckaert, B. De Sutter, and K. De Bosschere, "Hybrid static-dynamic attacks against software protection mechanisms." ACM Press, 2005, p. 75.
- [15] M. Plasmans, "White-box cryptography for digital content protection," Ph.D. dissertation, Technische Universiteit Eindhoven, 2005.
- [16] W. Speirs, "Dynamic cryptographic hash functions," Ph.D. dissertation, Purdue University, West Lafayette, IN, USA, 05 2007.
- [17] J.-P. Tillich and G. Zémor, "Hashing with sl_2 ," *Lecture Notes in Computer Science*, vol. 839, pp. 40–49, 1994.
- [18] V. Shpilrain, "Hashing with polynomials," in *Proceedings of ICISC 2006*. Springer, 2006, pp. 22–28.