Distributing Trust Verification to Increase Application Performance[*]

Mariano Ceccato[1], Jasvir Nagra[2], Paolo Tonella[1]
[1] Fondazione Bruno Kessler—IRST, Trento, Italy
[2] University of Trento, Italy
{ceccato, tonella}@fbk.eu, jas@dit.unitn.it

Abstract

The remote trust problem aims to address the issue of verifying the execution of a program running on an untrusted host which communicates regularly with a trusted server. One proposed solution to this problem relies on a centralized scheme using assertions and replication to withhold usable services from a tampered client. We show how to extend such a scheme to a distributed trusted hardware such as tamper-resistant smartcards. We compared the performance and security of the proposed distributed system to the original centralized scheme on a case study. Our results indicate that, compared to a centralized scheme, our distributed trust scheme has dramatically lower network traffic, and smaller memory and computational requirements on the trusted server.

1 Introduction

There are two key trust issues that arise in an open distributed computing setting. Firstly, a client that joins a distributed system may wish to be sure that the code it receives to execute has not been tampered and when executed it is not malicious. Secondly, a server would want to be sure that any computation it requests from a client is performed according to the conditions and rules prescribed by the server. The server is not willing to provide its services to untrusted clients, which may behave maliciously or unfairly with respect to the other clients.

While cryptographic signature schemes and sandboxed clients largely address the first of these concerns, the latter concern – that of software integrity – remains an open problem. The *Remote trusting* problem is a particular instance of the *software integrity* problem in which a trusted host (server) wishes to verify that an untrusted host (client) is executing according to its expectations at the point when the client requests a service.

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

A solution to the remote trusting problem is based on moving the tamper-sensitive part of the client computation to the server. Such a solution was investigated by Zhang and Gupta [13], in the context of protection from illegal software copying. Copy-sensitive parts of the clients are sliced and moved to the server so as to make copying ineffective, if based on the client code only. Another solution, investigated in [2], focuses on the notion of invalid state and exploits the barrier slicing technique to move the invalid-sensitive part of the client state to the server. Both solutions are based on the assumption that the core of trust is in the server and that every computation moved to the server becomes intrinsically safe.

The main problem of these two solutions based on slicing is that the server, which is the only reliable source of trust, is overloaded with computations that cannot be performed safely on the client. When the server is accessed by a high number of clients concurrently, it might be impossible to ensure an adequate quality of service.

In this paper, we propose a distributed trust architecture, which takes advantage of trusted hardware, such as smartcards, residing on each client. The core of trust is split between central trusted server and local trusted hardware, so as to delegate to the local trusted hardware everything that does not need a centralized service. The server keeps only its original services, which by design cannot be distributed, so that its load is unchanged, compared to the original application. Additional trust is achieved by means of the local trusted computations performed by the smartcards. The slices of trust-sensitive code are moved to the local trusted hardware.

2 Background

In this section we summarize the remote entrusting problem and the centralized solution. More details are available in the previous paper [2].

2.1 Remote trust verification

Remote trust verification involves a trusted host (server) S, an untrusted host (client) C and a communication channel between the two. The integrity of the application P running on C has to be verified whenever a communication act occurs between S and C.

We assume a scenario in which the application P requires a service delivered by S. To receive this service a communication channel is established between C and S and some messages are exchanged:

$$C[s] \xrightarrow{m} S \text{ and } S \xrightarrow{v} C[s]$$

where s is the current state of application P running on C and m is a message that requests some service from S. Once S receives the request m it replies by sending the message (service) v.

The state s of the client application during a communication with S is a valid state when it satisfies certain validity properties expressed through an assertion ($A(s) = \mathit{true}$). In order for S to trust the application P upon the execution of a communication act, P has to exhibit a valid state. The only way in which S can verify the validity of the application P is by analyzing the message m that C has sent. S *trusts* P upon execution of the communication act $C[s] \xrightarrow{m} S$ if $E(m) = \mathit{true}$, where E is an assertion checked by the server S.

Thus, the *remote trusting problem* consists of finding a protection scheme such that verifying $E(m)$ is equivalent to having a valid state (i.e., $E(m) \Leftrightarrow A(s)$).

A protection mechanism is not *sound* (attacker wins) whenever the server is trusting the client, but the current state of the client is not valid ($E(m) = \mathit{true} \;\wedge\; A(s) = \mathit{false}$). A protection mechanism is not *complete* when the server does not trust a client that should be trusted ($E(m) = \mathit{false} \;\wedge\; A(s) = \mathit{true}$).

In the attempt to break a given protection mechanism, we make the assumption that the attacker can: (1) reverse engineer and modify the code of P; (2) alter the running environment of P, for example through emulators or debuggers, and dynamically change the state of P; (3) produce static copies of P and execute multiple copies of P in parallel, some of which are possibly modified; and, (4) intercept and replace any network messages upon any communication act.

2.2 Using barrier slicing for remote trust

When the server S delivers a service v, the client C can make some use of v only if its state is consistent with the state in which the service was requested. More specifically, we can make the assumption that a (possibly empty) portion of the state s of C must be valid in order for the service v to be usable by the client. We call this substate of the client the *safe* substate. Formally, such substate $s_{|_{Safe}}$ is the projection of the state s on the safe variables of the client. Intuitively, when the service v is received in an invalid substate $s_{|_{Safe}}$, the application cannot continue its execution, in that something bad is going to happen (e.g., the computation diverges or blocks). The complement of the safe substate is called *unsafe* substate and is formally the projection of the state on the unsafe variables: $s_{|_{Unsafe}}$.

The intuitive idea behind the usage of barrier slicing for remote trusting is to move the portion (slice) of the application P that maintains the variables in $s_{|_{Unsafe}}$ to the server, in order to prevent the attacker from tampering with them. To limit the portion of code that needs to be moved, *barrier slicing* instead of regular slicing is used.

The regular, backward slice for a variable at a given program point (slicing *criterion*) gives the subprogram which computes the same value of that variable at the selected program point as the original program. Hence, slicing on the unsafe variables at the communication acts gives the portion of client code that, once moved to the server, ensures correct and reliable computation of $s_{|_{Unsafe}}$. Since it is the server which computes $s_{|_{Unsafe}}$, by executing one slice per client, no tampering can occur at all (the server is assumed to be completely trusted).

We can notice that the computation of the moved slices on the server is somehow redundant. In fact, such computation may involve values from $s_{|_{Safe}}$ which are ensured to be valid and thus do not need be recomputed on the server. In other words, whenever the computation carried out in a slice moved to the server involves a safe variable, it is possible to use the value obtained from the client (extracted from m), without any need to recompute it. The technique to achieve such a slice portion is called *barrier slicing* [4, 5], with the barriers represented by the safe variables which do not need to be recomputed. Technically, when including the dependencies in the backward transitive closure, the computation of the slice is stopped whenever a safe variable is reached.

3 Distributed trust verification

3.1 Centralized trust architecture

The barrier slicing solution to the remote trusting problem is based on a centralized architecture. The slice is moved onto the server and instructions are added to the client to cause the server execute the slice when needed. Data required to execute the slice is sent to the server by the client and required values are requested by the client when they are needed. This involves additional communication and synchronization, and requires the execution of a stateful computation (the moved slice) for each currently running client.

The additional communication and synchronization messages required in this scheme add a lot of network overhead to the application. Several optimizations were outlined in the original proposal [2] to reduce the number of communication messages required. In spite of these optimizations, in some applications the slowdown introduced by these additional communication acts may result in a unacceptably large slow down of the application.

The second performance penalty depends on the amount of computation that must be performed by the server. If the server is supposed to serve a high number of clients, the centralized solution [2] may not scale, overloading the trusted server with a large number of concurrent slice executions, which demand too much of computing power or memory. In fact, slice execution must be instantiated on a per-client basis and may involve substantial memory, since it is by definition a stateful computation.

3.2 Distributed trust architecture

In a distributed trust scheme, each client has a tamper-resistant computational device (e.g., smartcard) which is trusted by the server. In this work, we investigate a distributed trust architecture, used in conjunction with barrier slicing in order to achieve trusted computation of $s_{|_{Unsafe}}$ with minimal performance penalties. We make the assumption that the client is connected to some trusted, programmable hardware, such as a smartcard, possibly attached to the USB port. This hardware usually has limited computing and memory power. Hence, it is not possible to move the entire client application to it and solve the remote trusting problem in this way. However, the computing capabilities of such a device are probably sufficient to execute a barrier slice of the client, thus ensuring trusted computation of $s_{|_{Unsafe}}$.

In the rest of the paper we will use the term *smartcard* to generally refer to any tamper resistant hardware. The same protection mechanism can be implemented using any secure device, such as secure coprocessors, USB dongles or secure mobile devices.

Intuitively, the barrier slices that ensure remote trusting are moved and executed on the smartcard. The client continues to run the portion of application that is non-sensitive, from the security point of view (e.g., the GUI), or that is intrinsically secured (i.e., computation of $s_{|_{Safe}}$). The client continues to communicate with the server for any service v necessary to continue its execution. A virtual, secure channel is established between the smartcard and the trusted host, so that whenever the smartcard detects a deviation from the valid states, the server is notified and stops delivering its services to that particular client.

The program on client C in a state s sends a message $m = f(s)$ to the smartcard. Depending on the power of the smartcard, the smartcard either encrypts m using a key or verifies the assertion and encrypts a message to indicate the validity of m. The smartcard sends this encoded message $Enc(m)$ either directly or via the client to the server. These encoded messages serve as a secure virtual channel between the smartcard and the server. Based on this message, the server sends the client a service v. The assertion $E(m)$, checked either on the server or on the smartcard, determines whether the server should trust the client or not. The load on less powerful smartcards can be further reduced by allowing the client to send unencrypted messages m directly to the server. Instead, the smartcard periodically sends a checksum of recent messages to the server allowing the server to test the veracity of messages it has received since the last checksum, allowing it to detect tampered messages and withhold further service. In this scheme, there is a trade-off between the power required in the tamper-resistant hardware and length of time an attacker can successfully receive a tampered service before being detected by the server.

The architecture described herein addresses the two main performance penalties for the application running on the client. The network overhead is restored to levels comparable to the initial application. All communication acts necessary for the client to obtain the values of variables in $s_{|_{Unsafe}}$ do not transit over the network any longer. They go directly to the local smartcard. The computation and memory overhead are eliminated, since the barrier slices are not executed on the server any more. It is the smartcard that contains their code and runs them locally. Compared to the initial, untrusted architecture, the client and server pay minor penalties related to the authentication messages that are generated by the smartcard and transit over the network.

4 Program Transformation

As in the case of the centralized protection, the variables that need to be protected must be manually chosen by the user. Once these variables have been selected, the code to be moved to the smartcard can be automatically computed and the client code can be transformed. The transformed code is eventually installed on the client. In this section we cope only with code modifications that are required to apply the distributed protection mechanism. Other changes, required to make the code run on special hardware are out of the scope in this paper. Appropriate cross compilers or manufacturer proprietary tools should be used.

The transformation steps are described with reference to the example in Figure 1. Let us consider the fragment of code in Figure 1(a). If we consider the n-th communication, the barrier B_n is given by instruction 1, while the slicing criterion C_n is given by instructions 10 and 12. By computing

```
1     x = x * a;
2     a = a + x;
      send^h(m_h);
      receive^h(k_h);
3     a = x + a;
4     x = x + 1;
5     while (c) {
6        a = a + x;
7        x = x + a; }
8     x = x * a;
9     if (c)
10    then { a = 2 * x;
11       x = x + a;}
12    else { a = x * x;
13       x = x + 2*a; }
14    x = 2*a;
      send^n(m_n);
      receive^n(k_n);
```

(a)

```
C1    x = x * a;
C2    sync();
      send^h(m_h);
      receive^h(k_h);
C3    sync();
C4    x = x + 1;
C5    while (c) {
C6       sync();
C7       x = x + ask("a"); }
C8    x = x * ask("a");
C9    if (c)
C10   then { sync();
C11      x = x + ask("a");}
C12   else { sync();
C13      x = x + 2*ask("a"); }
C14   x = 2*ask("a");
      send^n(m_n);
      receive^n(k_n);
```

(b)

```
S1    a = a + x;
S2    sync();
      receive^h(m_h);
S3    x = m ;
S4    if A(x, a) then
         sendAuthenticityTag(m_h);
S5    else
S6       sendTamperedTag();
S7    a = x + a;
S8    sync();
S9    x = x + 1;
S10   while (c) {
S11      a = a + x;
S12      sync();
S13      x = x + a; }
S14   x = x * a;
S15   if (c)
S16   then { a = 2 * x;
S17      sync(); }
S18   else { a = x * x;
S19      sync(); }
      receive^n(m_n);
S20   x = m ;
S21   if A(x, a) then
         sendAuthenticityTag(m_n);
S22   else
S23      sendTamperedTag();
```

(c)

Figure 1. An example of the proposed protection scheme: (a) original client, (b) modified client and (c) corresponding smartcard.

the barrier slice, we obtain:

$$\mathit{Slice}_\sharp(C_n, B_n) = \{12, 10, 9, 8, 7, 6, 5, 4, 3, 2\}$$

4.1 Client code

The transformation of the client consists of removing some of the statements in the barrier slice and introducing some extra communication with the smartcard, to retrieve the needed values. The transformation is composed of the following steps:

- Every message m sent to the server is also sent to the smartcard.
- Every unsafe variable definition in the slice (Figure 1(a) statements 2, 3, 6, 10 and 12) is replaced by the instruction `sync()` (Figure 1(b) statements C2, C3, C6, C10 and C12). This message corresponds to a synchronous blocking communication, which means that the client has to wait for the answer from the smartcard. The smartcard sends an *ack* only when its execution reaches the corresponding `sync()` (Figure 1(c) statements S2, S8, S12, S17 and S19).
- Every use of variable `a` $\in$ Unsafe^n on the client is replaced by an `ask("a")` that requests the current value of `a` from the smartcard (Figure 1(b) statements 7, 8, 11, 13 and 14).
- The last change involves input values (i.e., user input, file read), which must be forwarded to the smartcard as soon as they are collected by the client application.
- On the client a new process is added (not shown in Figure 1(b)), that just waits for encrypted messages coming from the smartcard and forwards them as they are to the sever. In this way a virtual secure channel is established between the smartcard and the server.

4.2 Smartcard code

The code to be uploaded onto the smartcard aims at: (1) making the smartcard able to run the slice computing the *Unsafe* variables; (2) keeping it synchronized with the client; and, (3) verifying the validity of the *Safe* variables. The transformation of the client is composed of these steps:

- The very first change to apply is to copy the barrier slice $\mathit{Slice}_\sharp(C_n, B_n)$ to the smartcard. The smartcard has to boot strap the slice as the original client does (e.g., data structures must be initialized).
- The slice is fed with any input coming from the client.
- As soon as the smartcard receives a message m from the client, the validity of the client's state is verified (statements `S4-S6`, `S21-S23`), after extracting the values for the *Safe* variables (statements `S3`, `S20`). If the state is considered valid an *authenticity tag* is sent

to the server through the virtual secure channel. Otherwise, the server is notified about the identified attack, through a *tampered tag*. Tags are encrypted and signed using a secret key hidden in the smartcard, thus they are not visible to the attacker. They include the message m received from the client.

- Whenever a `sync()` statement is reached, the current values of the *Unsafe* variables are saved, after synchronizing with the client.

- A smartcard process (not shown in Figure 1(c)) replies to each client's `ask()` by sending the currently saved value for the requested variable.

Figure 1(c) shows the code running on on the smartcard after the transformation. Instruction 2 of the original application contains a definition of variable `a` $\in$ *Unsafe*. In the client, this instruction is replaced by a `sync()` (instruction `C2`), corresponding to the smartcard's `sync() S2`. Upon synchronization, when the client's execution is at `C2` and the smartcard's execution is at `S2`, the current value of the unsafe variable `a` is saved on the smartcard. The smartcard can then proceed until the next `sync()` (instruction `S8`), and any `ask()` issued by the client is replied by a parallel smartcard process sending the stored value of `a` (i.e., the value produced at `S1`). We can observe that instructions 11 and 13 are not duplicated on the smartcard, since they do not belong to the considered barrier slice.

4.3 Server code

In contrast to the centralized solution, the code of the server (not shown in Figure 1) is affected by only minor changes. A new process is added that establishes the virtual secure channel with the smartcard and waits for authenticity tags. In case they are not received at the expected rate, an invalid or a tampered tag is received, or the message m brought by the tag does not correspond to the message m received from the client, this process notifies the original server to stop delivering the service to the corresponding client. In alternate implementations where the assertion $E(m)$ is computationally expensive and the resources on the smartcard are meager, code to test the assertion is added to the server.

5 Experimental results

Both the distributed and centralized protection architectures have been applied on the same case study, a network application, and a scalability assessment has been performed on them.

5.1 Case Study

CarRace is a network game, the client of which consists of around 900 lines of Java code. The application allows players to connect to a central game server and race cars against each other. During the race, each player periodically sends data about the car position and direction to the server, which then broadcasts the data to the other clients allowing them to render the game on their screen. The fuel is constantly consumed, and a player must periodically stop the car and spend time refueling.

There are many ways a malicious user (the attacker) can tamper with this application and gain an unfair advantage over his competitors. For example, he can increase the speed over the permitted threshold, change the number of performed laps or avoid refueling by manipulating the fuel level. Unfortunately not all the variables that must be protected against attack are in *Safe*. The attacker cannot tamper with the position (variables `x` and `y`), because the displayed participants' positions are those broadcast by the server, not those available locally. The server can check the conformance of the position updates with the game rules (e.g., maximum speed). The other sensitive variables of the game (e.g., `gas`) are *Unsafe* and must be protected by some extra mechanism, such as barrier slicing. A barrier slice has been computed using the *Unsafe* variables as criteria and *Safe* variables as barrier. The barrier slice is quite small, just 120 lines of code (14% of the entire application) so it can fit low cost, commercially available, secure hardware, such as smartcards.

5.2 Scalability Analysis

The two different protection mechanisms (centralized and distributed architecture) have been applied to the case study application. In order to analyze how they scale when the number of connected clients increases, the server performance has been recorded in terms of memory consumption, number of threads and network traffic (i.e., exchanged messages). The server has been run respectively with 2, 4, 6, 8, 10 and 12 clients (the race is played pairwise). From the trend of the plotted data, the scalability of the system was assessed. In order to make the measurement objective and repeatable, a "softbot" was developed, able to play the race by driving the car through a sequence of check points.

Figure 2 shows the amount of memory (bytes) allocated on the heap of the Java Virtual Machine that runs the server. In the case of two clients, the memory required by the centralized solution is about the double of the distributed solution memory size (988 Vs 524 bytes). The slope of the line that connects the experimental points suggests the scalability of the two solutions. By linear fit, we can determine that while the centralized protection requires to allocate about

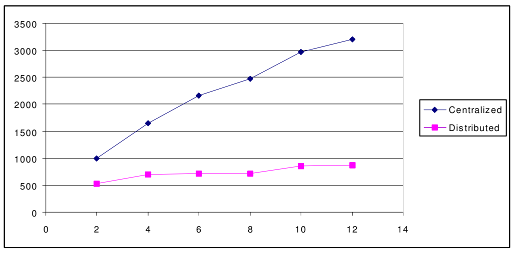

Figure 2. Memory consumption on the server.

220 bytes per each new client, the distributed one requires just 32 bytes per client (15%).

Results indicate that in the CarRace application, the slice executing on the smartcard consumed 820 bytes of heap memory compared to 2325 bytes on the original client. In other words, less than 40% of the memory in use in the original client ends up being required on the smartcard. Furthermore, the slice required less than 25% of the CPU cycles of the original application. While these are significant computational resources, they are a considerable improvement over using tamper-resistant hardware to protect the entire client application. The resources required to execute the smartcard slice vary greatly from application to application, so we expect higher or lower benefits depending on the given case. Optimizations are also possible for specific applications (see previous sections).

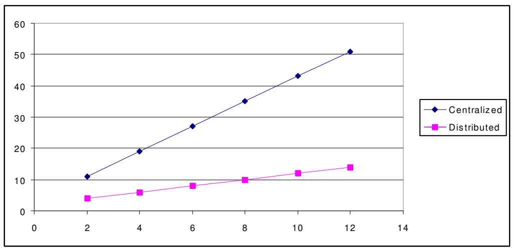

Figure 3. Number of threads on the server.

If we consider the threads running on the server, we can see on Figure 3 that the distributed solution is less demanding when the number of clients increases. In fact, while the centralized solution requires 4 new threads per served client, the distributed approach requires just one thread per client (25%). In fact, all the threads in the barrier slice must be replicated on the centralized server, for each connected client, while in the distributed solution just one more thread is necessary, to handle the encrypted network communication.

Figure 4 shows how network traffic (number of exchanged messages) changes on the server when the number

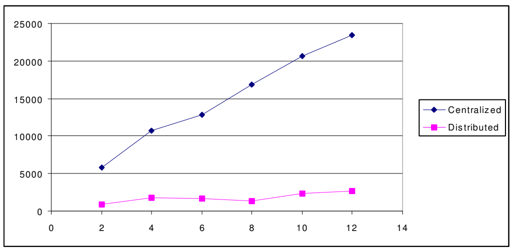

Figure 4. Number of exchanged messages.

of clients increases. In the distributed solution the number of messages exchanged by the server is the same as the original (i.e., non-protected) application, because only the original messages go through the server (145 messages per client). In fact, all the messages that keep the unsafe state updated go through the secure hardware (e.g., smartcard). In the centralized solution the server has to handle either the original messages and the security-related messages, causing the network traffic to increase faster (1743 messages per client, according to a linear fit). In the distributed solution, the server network traffic growth is about 8% of the centralized solution.

6 Related works

The problem of remote attestation of software has a colorful and long history. The key idea of a "trusted computing base"(TCB) can be traced to the Orange Book [7] and Lampson [6]. Lampson defines the TCB as a "small amount of software and hardware that security depends on". In this context, security was assured by the TCB because the operating system and hardware were assumed to be known, trusted and inviolable. More recently, trusted hardware schemes for remote attestation have been proposed. The Trusted Computing Group [8] and Microsoft's Palladium [1] have proposed several schemes based on a secured co-processor. These devices use physical defenses against tampering. The co-processor stores integrity measurement values that are later checked externally. The increased cost of manufacture and prohibitive loss of processing power due to the required cryptography has largely limited the mainstream adoption of these solutions.

Alternatives to *custom* trusted hardware are represented by software-only solutions that rely on *known* hardware. Swatt [10] and Pioneer [9] apply respectively to embedded devices and desktop computer. At run-time they compute a checksum of the in-memory program image to verify that no malicious modifications has occurred. They take advantage of an accurate knowledge of the client hardware and memory layout to precisely predict how long the checksum computation should take. Detection of tampering is based on an observed execution time that exceeds the upper bound, under the assumption that any attack introduces some levels of indirection, which increases the execution time.

Detailed client hardware knowledge is a reasonable assumption when a *collaborative* user is interested in protecting her/himself against malware. It does not apply to malicious users who are willing to tamper with the hardware and software configuration or provide incorrect information about it.

If checksum computation time can not be accurately predicted, the memory copy attack [12] can be implemented to circumvent verifications. A copy of the original program is kept by the malicious user. Authenticity verification retrieves the code to be checked in *data mode*, i.e., by means of proper procedures (*get code*) that return the program's code as if it were a program's datum. In any case, the accesses to the code in *execution mode* (i.e., control transfers to a given code segment, such as method calls) are easily distinguished from the accesses in *data mode*. Hence, the attacker can easily redirect every access in execution mode to the tampered code and every access in data mode to the original code, paying just a very small performance overhead.

Kennell and Jamieson [3] propose a scheme called Genuinity which addresses this shortcoming of checksum-based protections by integrating the test for the "genuineness" of the hardware of the remote machine with the test for the integrity of the software that is being executed. Their scheme addresses the redirection problem outlined above by incorporating the side-effects of the instructions executed during the checksum procedure itself into computed checksum. The authors suggest that the attackers only remaining option, simulation, cannot be carried out sufficiently quickly to remain undetected. Shankar et al. [11] propose two substitution attacks against Genuinity which exploit the ability of an attacker to add code to an unused portion of a code page without any additional irreversible side-effects.

7 Conclusions

In this paper we address the remote trusting problem, verifying the healthy execution of a given application that runs on a untrusted host. Our solution consists of a distributed architecture. The application to protect is divided into two segments using barrier slicing. The portion that keeps sensitive variables up to date is moved to local, secure hardware, in order to protect it against tampering.

This solution represents an improvement of the previous centralized solution, proposed in [2], where the sensitive application part is moved to the server. On a small case study, we observed that the centralized architecture causes unacceptable server overhead, when many clients are connected. The distributed solution has better scalability. It requires considerable less server resources in terms of allocated memory (15%), threads (25%) and network traffic (8%), while providing the same level of protection. On the other hand, the portion of code to move onto the local, secure hardware (14% of the total application) is small enough to fit a smartcard.

References

[1] A. Carroll, M. Juarez, J. Polk, and T. Leininger. Microsoft "Palladium": A Business Overview. *Microsoft Content Security Business Unit, August*, 2002.

[2] M. Ceccato, M. D. Preda, J. Nagra, C. Collberg, and P. Tonella. Barrier slicing for remote software trusting. In *Proc. of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 27–36. IEEE Computer Society, Sept. 30 2007-Oct. 1 2007.

[3] R. Kennell and L. H. Jamieson. Establishing the genuinity of remote computer systems. In *Proceedings of 12th USENIX Security Symposium*, 2003.

[4] J. Krinke. Barrier slicing and chopping. In *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 81–87, 2003.

[5] J. Krinke. Slicing, chopping, and path conditions with barriers. *Software Quality Journal*, 12(4):339–360, dec 2004.

[6] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: theory and practice. *ACM Trans. Comput. Syst.*, 10(4):265–310, 1992.

[7] D. of Defense. Trusted computer security evaluation criteria, dod 5200.28-std. Washington D.C., December 1985. DOD 5200.28-STD.

[8] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proceedings of the 13th USENIX Security Symposium*, pages 223–238, 2004.

[9] 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.

[10] A. Seshadri, A. Perrig, L. van Doorn, and P. K. Khosla. Swatt: Software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy*, pages 272–283, 2004.

[11] M. C. Umesh Shankar and J. D. Tygar. Side effects are not sufficient to authenticate software. Technical Report UCB/CSD-04-1363, EECS Department, University of California, Berkeley, 2004.

[12] P. van Oorschot, A. Somayaji, and G. Wurster. Hardware-assisted circumvention of self-hashing software tamper resistance. *IEEE Transactions on Dependable and Secure Computing*, 2(2):82–92, April-June 2005.

[13] X. Zhang and R. Gupta. Hiding program slices for software security. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 325–336, Washington, DC, USA, 2003. IEEE Computer Society.