Seventh IEEE International Working Conference on Source Code Analysis and Manipulation

Barrier Slicing for Remote Software Trusting*

Mariano Ceccato[1], Mila Dalla Preda[2], Jasvir Nagra[2],
Christian Collberg[3], Paolo Tonella[1]

[1] Fondazione Bruno Kessler—IRST, Trento, Italy

[2] University of Trento, Italy [3] University of Arizona, USA

Abstract

Remote trusting aims at verifying the "healthy" execution of a program running on an untrusted client that communicates with a trusted server via network connection. After giving a formal definition of the remote trusting problem and a test to determine whether an attack against a given remote trusting scheme is successful or not, we propose a protection against malicious modification of the client code, based on the replication of a portion of the client on the server. To minimize the size of the code that is replicated, we propose to use barrier slicing. We show the feasibility of our approach on a case study. Our results indicate that a barrier slice is significantly smaller than the corresponding backward slice while providing the same level of protection.

1 Introduction

The *Remote trusting* problem is a particular instance of the *software integrity* problem. In software integrity, the problem is to ensure that a given program is executed unmodified—verifying in this way that the program has not been tampered with. In remote trusting, the problem is to ensure that a given program running on an untrusted host (client) is executing according to the expectations of the trusted host (server), but only when the two communicate over the network (e.g., during service delivery).

The most significant issue in remote trusting is that the trusting party (server) has no control over the untrusted party (client). The server can not rely on the client hardware configuration, for example, to predict the execution time of the original program in order to detect an execution delay that can be due to malicious modifications. The hardware configuration can not be considered known because the client user could lie about it. The client user can not be considered a collaborative user, he/she could be interested in tampering with the client software to make the application work differently than expected: the user of the client application is not trusted, in that he/she could gain some benefits by running a tampered application (e.g., paying a reduced fee).

The attacker can take advantage of any dynamic and static program analysis tool to reverse-engineer the application. He/she can directly modify the application code or install simulation and debugging environments to tamper with the execution. On the other hand, the server is willing to communicate only with clients that have not been tampered with. The server is expected to deliver a certain service only to genuine clients; modified clients should be detected and refused. Remote trusting can be applied to all those applications that need the network to work properly, for example because they need a service delivered by a server (e.g., Internet games). Before deciding whether to deliver the requested service or not, the server may want the application requesting the service to prove that it has not been tampered with by a malicious user.

In this paper we propose a solution to the remote trusting problem, based on the observation that a portion of the client can be easily verified to be sane through assertions. In fact, some of the services delivered to the client are unusable if the client's state does not match the server's assumptions, expressed through assertions. However, in general this mechanism does not provide protection for the whole sub-state of the client that the server wants to rely on. We propose to use program slicing to identify the remaining portion of the client that can not be verified through assertions, but that is still sensitive. The idea is to move this relevant part of the application from the client to the server, so it can be run untampered. A similar approach was used by Zhang and Gupta in order to prevent software piracy [13]. Their idea was to turn a stand-alone application into a network application by moving a relevant slice of the application to the server. The criteria used to determine the fragments of code that reside on the server and on the client ensure that it is difficult for an attacker to recover the original application, preventing in this way illegal copying.

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

0-7695-2880-5/07 $25.00 © 2007 IEEE
DOI 10.1109/SCAM.2007.27

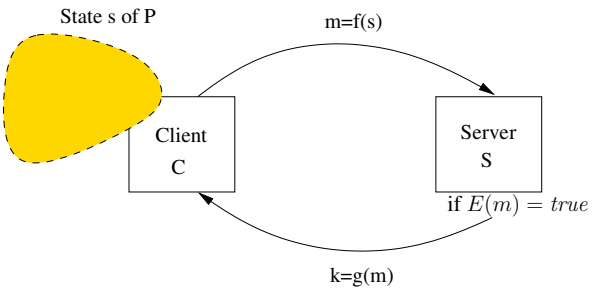

Figure 1. Overview of remote trusting.

In order to reduce the size of the computation to be moved to the server, we take advantage of the client's substate secured through assertions. The values of the variables secured in this way can act as *barriers* and computation of the transitive closure of program dependencies can stop at such barriers, since the server knows these values and is sure that they can be trusted. After computing the barrier slices, program transformations are applied to generate the secured client and the corresponding server.

In Section 2 the remote trusting problem is formally defined. After describing the existing solutions in Section 3, our approach is presented: the usage of barrier slicing is presented in Section 4 and program transformations are presented in Section 5. Our method is then applied to a case study in Section 6, where the results are discussed. Conclusions and future works close the paper in Section 7.

2 Problem definition

Remote trusting focuses on *network applications*, i.e., applications that need to access services provided by other machines over the network. Thus, the remote trusting scenario consists of a service provider (*server*) and a service consumer (*client*), with the former running on the trusted machine and the latter on the untrusted one. The server is willing to deliver its services only to clients that are in a valid state and can be trusted.

An example of network application falling in the scope of remote trusting is the implementation of the TCP/IP stack protocol. In this case, a client is in a valid state if it obeys the policies enforced to avoid and rapidly solve network congestion. Another suitable example is an on-line computer game client. A healthy execution is one that does not result in any unfair advantage for those users that run a hacked version of the game.

2.1 Remote trusting scenario

The remote trusting scenario is shown in Figure 1. S is the trusted host (server) and C is the untrusted host (client) running a certain application P, whose integrity has to be verified upon communication with S. 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:

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

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) k. In general we have that:

- Message m depends on the current state s of application P, i.e., $m = f(s)$, where f is a function that converts a state into a message that can be understood by the server S.
- Message k depends on the previous message m, i.e., $k = g(m) = g(f(s))$, where g is a function that given a message sent by the client returns a message containing the service needed by the application.

So far, we have focused on a single communication act. However, in a real scenario, a sequence of communication acts is expected to occur. The assumptions and definitions in this paper apply to each of them: we are implicitly quantifying over each communication act in a sequence, even when the sequence index does not appear explicitly.

2.2 Problem definition

The current state of the client application P during communication with S is a valid state when it satisfies certain validity properties expressed through an assertion A.

Definition 1 Application P is in a *valid state* s upon execution of the communication act $C[s] \xrightarrow{m} S$ if $A(s) = true$, where A is an assertion.

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.

Definition 2 S *trusts* P upon execution of the communication act $C[s] \xrightarrow{m} S$ if $E(m) = true$, where E is an assertion.

Thus, the *remote trusting problem* consists of finding a protection scheme such that:

$$E(m) \Leftrightarrow A(s) \tag{1}$$

upon execution of $C[s] \xrightarrow{m} S$ and $S \xrightarrow{k} C[s]$. The server trusts a client if and only if it is in a valid state. When

condition (1) is satisfied, the server is able to detect any attack that compromises the validity of the application state during communication. The protection mechanism is not *sound* (attacker wins) whenever the server is trusting the client, but the current state of the client is not valid. Namely when there exists a communication act $C[s] \xrightarrow{m} S$ such that:

$$E(m) = true \quad \wedge \quad A(s) = false \tag{2}$$

We can observe that a server can trivially avoid this situation by refusing to trust any client, i.e., $E(m) = false$ for every m. However, for a protection mechanism to be useful, a server must trust application P running on the client whenever it is in a valid state. In fact, a protection scheme is not *complete* when there exists a communication act $C[s] \xrightarrow{m} S$ such that:

$$E(m) = false \quad \wedge \quad A(s) = true \tag{3}$$

This is the reason for the double implication in condition (1).

2.3 Attack model

In our framework the attacker is anyone who may want to alter the application's state, either dynamically or statically, to gain personal advantage in a forbidden way. The attacker has no restriction on the tools and techniques to use to reverse-engineer and then to tamper with the application (e.g., super-user privileges are assumed to be available to the attacker). He/she can install any software on the client machine (e.g., debuggers, emulators). The attacker can read and write every memory location, processor registers and files. Network traffic and operating system are fully visible and changeable for the attacker. Moreover the attacker can start the malicious activity at any time, not just when the application is running.

Even if the attacker can do almost everything on the client, he/she has no access/visibility on the server. The attacker neither knows what software is running nor what is the underlying hardware and operating system. The server is considered completely trusted, so no tampering can happen on it and no external view on its internal details is visible to an external observer.

The possible attacks can be grouped into four classes:

1. Reverse-engineering and modification of the code of P;
2. Modification of the running environment of P, for example through emulators or debuggers, and dynamic change of (part of) the state of P, without actually changing the code of P;
3. Production of static copies of P and execution of multiple copies of P in parallel, some of which are possibly modified;
4. Interception and replacement of network messages upon any communication act.

In order to detect attacks in class 1 alone, the verification of the static properties, such as code checksum, could be enough. However, if we consider also class 2 and class 3, this is not enough, because while running the tampered code the attacker could keep a correct program copy and use it to compute the correct checksum when required.

Also the verification of dynamic properties does not represent a strong protection. Attacks in classes 3 and 4 can redirect any dynamic check to the correct execution of a program clone, while actually running and making the server serve the tampered copy.

3 Existing solutions

The problem of remote attestation of software has a colorful history. The key idea of a "trusted computing base"(TCB) can be traced to the Orange Book [6] and Lampson [5]. 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 [7] 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 contains a private key, trusted code must be signed and the signature verified by the secure co-processor before code is executed. The increased cost of manufacturing and prohibitive loss of processing power to the cryptography required 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 [9] and Pioneer [8] apply 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 have occurred. They take advantage of an accurate knowledge of the client hardware and memory layout so as to be able to precisely predict how long the checksum computation should take. It is assumed that any attack introduces some indirection (e.g. redirecting memory checksum to a correct copy of the current program while a tampered copy is running). This indirection increases the execution time and thus can be used to detect tampering.

In the remote trust scenario, it is unreasonable to assume a *collaborative* user or detailed knowledge of the hardware. A malicious user may be 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 [11] 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 small performance overhead.

Kennell and Jamieson [2] 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. [10] 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.

Our solution is completely different from the previous ones, in that it does not rely on any hardware or any precise time computation, which is hard to achieve in the presence of non-collaborative users. We propose to use barrier slicing and program transformations to ensure that the critical portion of the client computation that cannot be protected through assertions is executed on the server.

4 Barrier slicing

A (backward) slice [12] on a given criterion (i.e., a variable at a given statement) is a sub-program that is equivalent to the original program with respect to the given criterion (assuming termination). Intuitively, the slice contains all the statements that affect the value of the variable in the criterion.

A slice can be computed as the transitive backward closure of data and control dependencies, resulting in all the statements on which the criterion depends directly or indirectly. A barrier slice [3, 4] is a slice computed on a code where some special statements are marked as barriers, meaning that they involve computations that are considered not to belong to the slice (e.g., because they are uninteresting or because the values of the involved variables are known). Barrier slices can be computed by stopping the computation of the transitive closure of the program dependencies whenever a barrier is reached.

Given the program dependency graph (PDG): (N, E), the (backward) slice with criteria $C \subseteq N$ and with barrier $B \subseteq N$ can be computed as:

$$Slice_\sharp(C,B) = \left\{ m \in N \middle| \begin{array}{l} p \in m \longrightarrow^* n \ \wedge \ n \in C \ \wedge \\ p\,\langle n_1 ... n_l \rangle : \\ \forall 1 \leq i \leq l : \ n_i \notin B \end{array} \right\}$$

where $p \in m \longrightarrow^* n$ denotes a path in the graph from m to n.

4.1 State partitioning

Given a program P let Var be the set of variables occurring in P. A program *state* s is a map $s : Var \rightarrow Values$ that associates a value with each variable in P. Given a subset $X \subseteq Var$ of variables, let $s_{|_X}$ denote the restriction of state s on X, i.e., $s_{|_X} : X \rightarrow Values$ where $\forall x \in X : \ s_{|_X}(x) = s(x)$. In this case we say that $s_{|_X}$ is a *substate* of s.

Let us consider the service k delivered by the server S to the client C during communication $S \xrightarrow{k} C[s]$. The *usability* of message k from application P running on the client depends on a substate of s. Intuitively, when the service k is received in an invalid substate, the application cannot continue its execution, in that something bad is going to happen (e.g., the computation diverges or blocks).

Let $Safe \subseteq Var$ be the subset of program variables that determines the usability of message k, and let $Unsafe = Var \setminus Safe$. This means that $s_{|_{Safe}} : Safe \rightarrow Values$ is the substate of s responsible for the usability of message k. Moreover, let us assume that the assertion A on state s can be decomposed as follows: $A(s) = A_{Safe}(s_{|_{Safe}}) \wedge A_{Unsafe}(s_{|_{Unsafe}})$.

Definition 3 Let k be a message generated by a valid state s, i.e., $k = g(f(s))$ with $A(s) = true$. k is *usable* by a different state $\hat{s}$ upon execution of the communication act $S \xrightarrow{g(f(s))} C[\hat{s}]$ if:

$$A_{Safe}(\hat{s}_{|_{Safe}}) = true$$

We have two possible cases (see Figure 2):

1. $Unsafe = \varnothing$. In this case the remote trusting problem can be trivially solved by choosing $m = s$ and $E = A$. In fact, this ensures that whenever the current state $\hat{s}$ of the client C is not valid, i.e., $A(\hat{s}) = false$, even if the attacker sends a valid state s to the server, i.e., $A(s) = true$, the service provided by the server cannot be used by the application.

2. $Unsafe \neq \varnothing$. In this case the above solution cannot be applied. In fact, an attacker could send a valid state s,

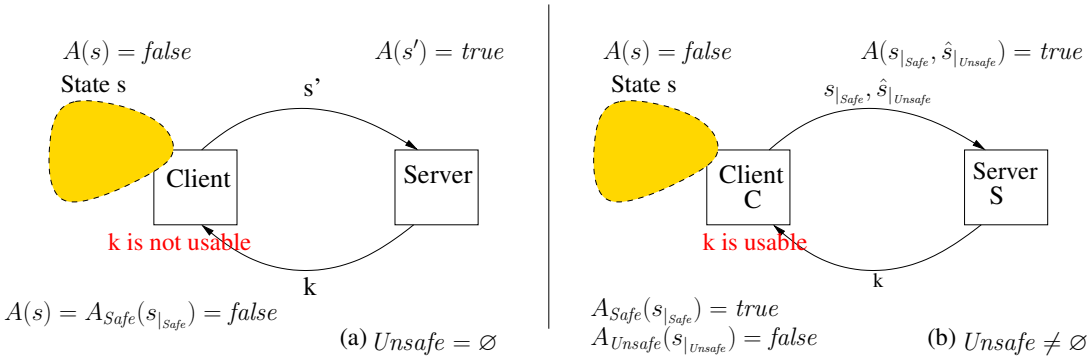

Figure 2. In (a) $Unsafe = \emptyset$ and the trivial solution can be applied. In (b) $Unsafe \neq \varnothing$ and an additional mechanism is required for the server to the values of variables in $Unsafe$ on the client.

i.e., $A(s) = true$, to the server while the current state is $\hat{s} = (s_{|_{Safe}}, \hat{s}_{|_{Unsafe}})$ such that $A_{Safe}(s_{|_{Safe}}) = true$ while $A(\hat{s}) = false$. The service $g(f(s))$ is usable, since $A_{Safe}(s_{|_{Safe}}) = true$, but the overall state $\hat{s}$ is invalid, making the protection $m = s$ and $E = A$ fail. It is clear that in order to perform this attack the attacker needs to reverse-engineer the application and identify the two substates $s_{|_{Safe}}$ and $s_{|_{Unsafe}}$ upon each communication act.

After the sensitive variables have been classified as *Safe* and *Unsafe*, barrier slicing can be effectively used to develop a protection scheme that works in case (2).

4.2 Barrier slicing to protect unsafe substate

The core idea of our proposed solution is to move the portion (slice) of application P that maintains the variables in *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 we will use *barrier slicing* instead of regular slicing. The proposed change implies some extra communication between the server and the client, because whenever the client requires the value of a variable in *Unsafe*, the server must provide it. In order to limit the communication overhead, some of the instructions that are moved to the server are maintained also on the client (these instructions are actually duplicated).

Let us identify each communication act by a number $n \in \mathbb{N}$ and correspondingly partition the variables into $Safe^n$ and $Unsafe^n$. **send**^{n} and **receive**^{n} denote respectively the send, i.e., $C[s] \xrightarrow{m} S$, and receive, i.e., $S \xrightarrow{m} C[s]$, during the n-th communication. On the server side, the statement **send**^{n} provides reliable values (i.e., *defines*) for all variables in $Safe^n$ that can be obtained from m_n. In PDG all those statements that consist in a communication

```
1    x = x * a;
2    a = a + x;
     send^h(m_h = x);
     receive^h(k_h);
     ------------------------------------
     Safe^h = {x},   Unsafe^h = {a}
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 = x);
     receive^n(k_n);
     ------------------------------------
     Safe^n = {x},   Unsafe^n = {a}
```

Figure 3. Fragment of the client application with two subsequent communication acts.

act are annotated with the corresponding number n. When there exists a path in the annotated PDG that connects the h-th send to the n-th send we say that the h-th communication precedes the n-th one, denoted $h \preceq n$.

Let us consider, for example, the fragment of the application P running on the client C in Figure 3. The h-th communication act precedes the n-th one. At the n-th communication act, we need to protect the backward slice of the definitions of variables in $Unsafe^n$ that might reach **send**$^n(\mathtt{m}_n)$ (i.e., a at statements 10 and 12 in our example). While computing the backward slice, we can halt when we encounter a statement that defines a variable that belongs to $Safe^h$,

with $h \preceq n$, and that might reach **send**$^h(\mathtt{m}_h)$, for example, statement 1 in Figure 3. This means that we are computing the barrier slice for the computation of the unsafe variables with the barrier given by the statements that produce valid values of variables communicated to the server during previous communication acts.

Thus, the barrier B_n for the n-th communication is given by the set of statements that modify the values of variables in $Safe^h$ such that the h-th communication precedes the n-th communication, formally :

$$B_n \stackrel{\text{def}}{=} \left\{\text{reach-def}(\mathtt{x}, \textbf{send}^h) \,\middle|\, \mathtt{x} \in Safe^h, h \preceq n \right\}$$

where reach-def(x, n) denotes the set of nodes corresponding to the definitions of variable x that might reach node n. We further assume that **send**^{h} is a postdominator of reach-def$(\mathtt{x}, \textbf{send}^h)$, i.e., whatever definition holds, it is necessarily transmitted to the server.

In fact, B_n precisely corresponds to the set of statements that are protected by the previous communication acts. With reference to the example in Figure 3, the barrier contains just statement 1.

The slicing criterion C_n for the n-th communication act is given by:

$$C_n \stackrel{\text{def}}{=} \left\{\text{reach-def}(\mathtt{a}, \textbf{send}^n) \,\middle|\, \mathtt{a} \in Unsafe^n \right\}$$

The computation of every unsafe variable whose value can reach the communication act must be moved to the server. In our example, the computation of `a` at statements 10, 12.

5 Program transformation

Once the barrier slice has been computed, the client and the server code can be transformed automatically in order to implement the proposed protection mechanism. The transformation steps are described with reference to the example in Figure 4.

Let us transform the fragment of code in Figure 4(a). If we consider the n-th communication, the barrier B_n is given by instruction 1, while the criteria C_n is given by instructions 10 and 12. By computing the barrier slice, we obtain:

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

5.1 Client side changes

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

- Every unsafe variable definition in the slice (Figure 4(a) statements 2, 3, 6, 10 and 12) is replaced by the instruction `sync()` (Figure 4(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 server. The server sends an ack only when its execution reaches the corresponding `sync()` (Figure 4(c) statements S2, S8, S12, S17 and S19).
- Every use of variable $\mathtt{a} \in Unsafe^n$ on the client is replaced by an `ask("a")` that requests the current value of `a` from the server (Figure 4(a) 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 server as soon as they are collected by the client application.

5.2 Server side changes

The transformation on the server side aims at: (1) making the server able to run the slice computing the *Unsafe* variables; (2) keeping it synchronized with the remote client; and, (3) verifying the validity of the *Safe* variables. It is composed of these steps:

- The very first change to apply is to copy the barrier slice $Slice_\sharp(C_n, B_n)$ to the server. The server has to boot strap the slice as the original client does (e.g., data structures must be initialized). One slice is run for each served client.
- The slice is fed with any input coming from the client.
- As soon as the server 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`).
- Whenever a `sync()` statement is reached, the current values of the *Unsafe* variables are saved, after synchronizing with the client.
- A server process (not shown in Figure 4) replies to each client's `ask()` by sending the currently saved value for the requested variable.

Figure 4(c) shows the code running on on the server after the transformation. Instruction 2 of the original application contains a definition of variable $\mathtt{a} \in Unsafe$. In the client, this instruction is replaced by a `sync()` (instruction `C2`), corresponding to the server's `sync() S2`. Upon synchronization, when the client's execution is at `C2` and the server's execution is at `S2`, the current value of the unsafe

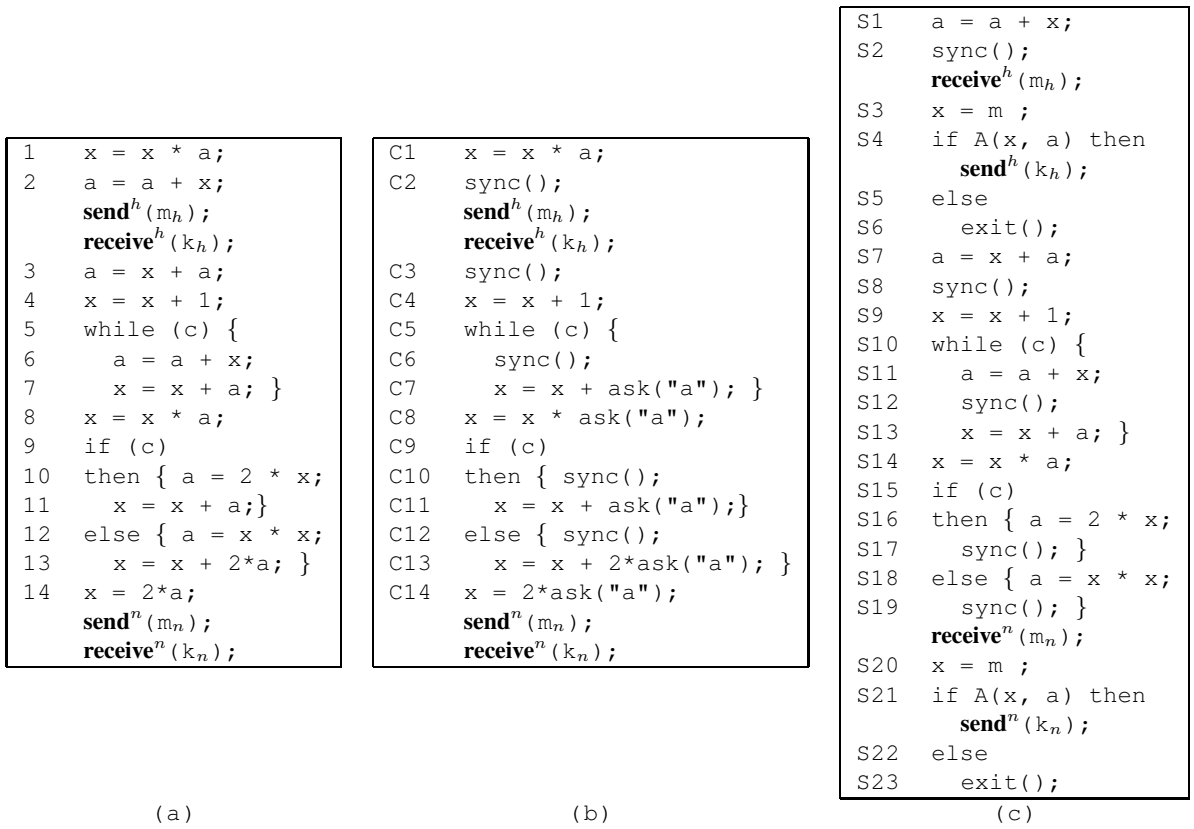

Figure 4. An example of the proposed protection scheme: (a) original client, (b) modified client and (c) corresponding server.

variable `a` is saved on the server side. The server can then proceed until the next `sync()` (instruction `S8`), and any `ask()` issued by the client is replied by a parallel server 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 server, since they do not belong to the considered barrier slice.

If a complex data structure that involves pointers, such as a linked list, is either only on the client (not involved in the slice) or only on the server (completely sliced out) the current solution works correctly. Otherwise, if the same pointers must be accessed both by the client and by the server, we should cope with the fact that actual pointer values could differ between the two hosts. We do not address directly this problem, because we made the assumption (valid in languages like Java) that all the pointers are not modified directly, but data structures are only accessed through proper handler (e.g., insert, delete and search methods). Thus, even if pointer values are different, structures should be consistent between the two hosts.

5.3 Optimizations

5.3.1 Removing control statements

After removing the definitions of the *Unsafe* variable from the client, some variables may became dead (no longer used). Dead variables can be easily identified through static analysis and removed from the client code. When the body of a loop or a branch of a conditional statement end up containing only `sync()` instructions, they can be deleted altogether from the client code, and the `sync()` can be moved afterwards both on the client and on the server.

Removing loop and conditional statements is potentially important, in that it may result in a dramatic decrease of the number of synchronization statements. In Figure 5(a) a portion of computational code has been subjected to the proposed transformation, where x_1 is in *Safe* while a_1 and a_2 are in *Unsafe*. In Figure 5(b) every use of unsafe variables is replaced by an `ask()`, and every definition of them is replaced by a `sync()`.

Some *if* statements can be removed immediately (lines 7-8 and 15-16). Two `sync()` calls replace them. After

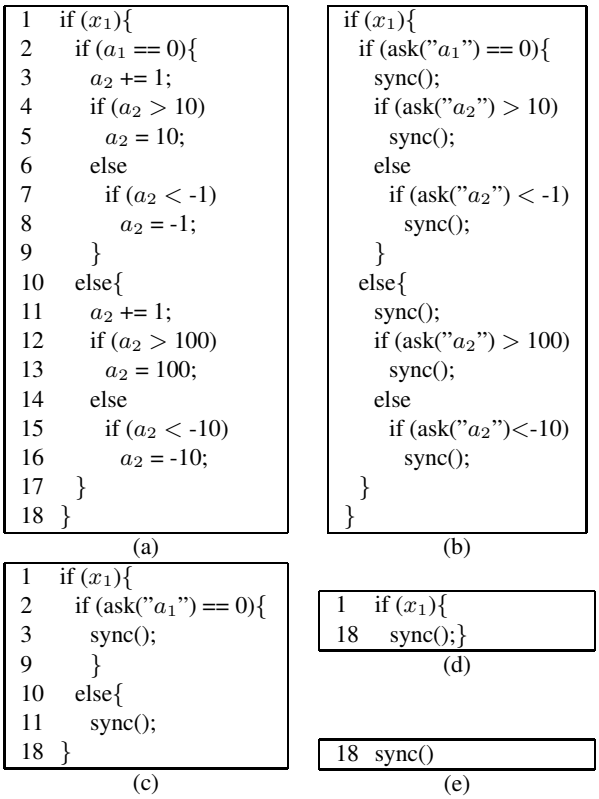

Figure 5. An example of the proposed protection scheme with optimization, (a) original code (b) transformed code, (c) (d) different steps in the optimization and (e) final code.

that, two more *if* statements can be removed (4 and 12). The resulting code is shown if Figure 5(c), with consecutive `sync()` instructions replaced by single `sync()`. Now the optimization can be re-applied and statements 2, 3, 9, 10, 11 can be removed and replaced by a single `sync()`. The result is shown in Figure 5(d) where just the the top-level *if* remains with just one `sync()` instruction inside. This *if* statement can be also removed and the `sync()` instruction is the only one that remains (Figure 5(e)).

Whenever a control statement is removed and replaced by a `sync()` on the client, the same operation must be replicated on the server's code, so as to ensure proper synchronization between the two. For example, when statements `7−8` are removed and replaced by a sync(), the corresponding *if* statement must be located on the server's code and the contained sync() must be moved immediately after it. If no other statement is inside the *if*, it is possible to remove it also from the server, but in general this may be not the case.

5.3.2 Pre-caching

The proposed solution may degrade the application's performance, due to the need for additional synchronizations. One way to limit this performance loss is to store the successive updates of the variables in $Unsafe$ on the server, making all the synchronization messages unnecessary. When the client needs the value of an *Unsafe* variable, it will ask the server for the i-th update of the variable. In particular:

- For every variable `a` $\in Unsafe^n$ we define a counter $\texttt{count}_\texttt{a}$ which is initialized to 0 both on the client and on the server.
- Every definition of variable `a` $\in Unsafe^n$ is replaced by $\texttt{count}_\texttt{a}$`++` on the client, while on the server every definition of variable `a` $\in Unsafe^n$ is followed by an increment of the counter ($\texttt{count}_\texttt{a}$`++`) and by a store of the updated value of variable `a` (`store("a",`$\texttt{count}_\texttt{a}$`, a)`).
- Every use of variable `a` $\in Unsafe^n$ on the client is replaced by an `ask("a",` $\texttt{count}_\texttt{a}$`)` that requires the current value of `a` from the server.

The result of applying this transformation to the previous example is shown in Figure 6.

Potentially, the variable $\texttt{count}_\texttt{a}$ could still be tampered with but, in practice, an attacker can not gain any advantage through that. Wrong values of $\texttt{count}_\texttt{a}$ will always be mapped to valid states maintained by the server.

The two optimizations are mutually exclusive, the first saves memory on the server, but it requires to exchange more messages, whereas the second one requires less communication but it consumes more server memory. It is up to the developer to analyze the context and chose which one to apply, considering also that the second optimization is not appropriate when the slice involves big data structures that would be kept in multiple copies.

6 Experimental results

The proposed protection mechanism has been applied to a case study application. We report the results and discuss them in this section.

6.1 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 server, which then broadcasts the data to the other clients allowing

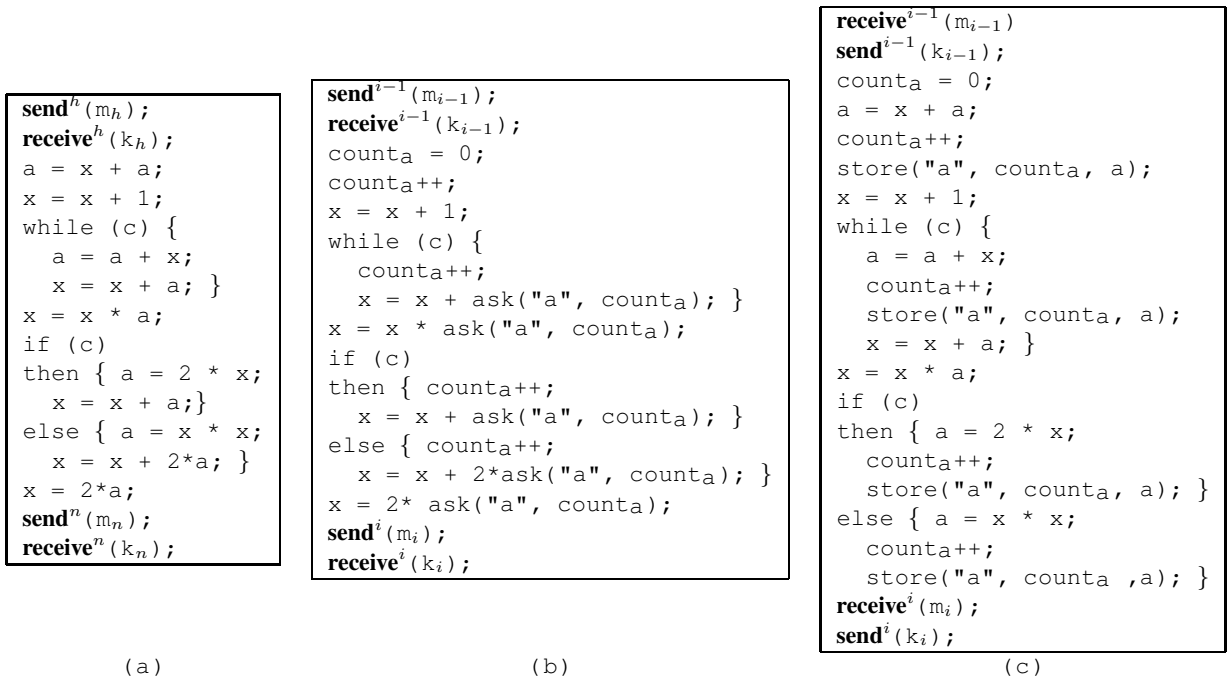

Figure 6. An example of the proposed protection scheme with counter, (a) original and (b) modified client, (c) corresponding server.

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 can tamper with this application in order to gain an unfair advantage over his competitors. For example, he can increase 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.

6.2 Slice size

Both backward slicing and backward barrier slicing were computed on the case study code, using the only `send()` in the program both as the criterion and as the barrier, as described in Section 4. Table 1 reports the size of the slices compared to the original program. The last column compares the size of the barrier slice with the regular slice.

	Original client	Slice	Barrier slice
LoC	858	185	120 (-65)
		22%	14% (-35%)

Table 1. Size of the backward slice and barrier slice compared to the original client.

The code that must be replicated on the server to protect *Unsafe* is small, both using regular and barrier slicing (respectively 22% and 14% of the total). However, the barrier slice (120 lines) is considerable smaller than the regular slice (185 lines).

Manual inspection of the code reveals that non-sensitive code consists mainly of functionalities that build the graphical user interface, handle graphical events, communicate over the network, handle the game message protocol and manage the data for the opponent cars. This explains the remarkable size reduction achieved through barrier slicing.

6.3 Performance

The proposed code transformation has been applied to the case study in two variants, the plain solution and the optimized solution. A full race was played and performance

Standard			
	Regular messages	**Trust messages**	**Increase**
Sent	1142	6796	5.95
Received	1144	6796	5.94
Optimized			
	Regular messages	**Trust messages**	**Increase**
Sent	1174	5910	5.03
Received	1172	5910	5.04

Table 2. Performance in terms of exchanged messages in the protected application.

was evaluated using the two versions. The user noticed just a very small delay between the commands and the car response in the non-optimized run. In the second, optimized run (see Subsection 5.3.1), no noticeable difference was observed, compared to the original game.

We also measured the communication overhead involved in the synchronization of the client with the barrier slice executed on the server. Table 2 shows the number of messages exchanged. *Regular messages* required by the original code are in the second column, whereas the third column reports the number of *trust* messages that are required by the protection mechanism (`sync`, `ask` and *input-forward* messages). The last column shows the increase, which is around 6 in the non-optimized version, and drops down to around 5 in the optimized one. The size of the two kinds of messages are similar (regular and trust messages are respectively about 150 bytes and 100 bytes long).

Although the amount of exchanged trust messages is remarkable in both versions, this did not result in a perceived performance degradation. One possible explanation is that this application was developed as a multi-thread application, so that some threads may continue to run while others are blocked, waiting to synchronize with the server.

7 Conclusion and Future Work

In this paper, we addressed the problem of remote trusting, i.e., verifying the healthy execution of a given application on a remote client before delivering a service to it. Our proposed solution relies on barrier slicing to identify which portions of the client code should be moved to the server in order to protect otherwise unsafe variables.

A preliminary study of the feasibility of this solution was based on CarRace, a network game written in Java. The barrier slice moved to the server was small with respect to the entire application (14%) and substantially smaller than a regular backward slice (35% less statements). The communication overhead increased by a factor of 5 or 6 (depending on the level of optimization), but the perceived performance did not degrade.

In future, we will consider larger case studies, to evaluate how the approach scales. In contrast to the current approach, we will also investigate automatic identification of *Safe* and *Unsafe* variables and the possibility of combining our approach with code obfuscation. Moreover, we intend to consider a trust scenario where multiple clients interact directly with one another (in a peer-to-peer way) and with the server, taking advantage of the information gathered by the clients on the peers connected to them.

References

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

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

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

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

[5] 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.

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

[7] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. pages 223–238, 2004.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] M. D. Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. PhD dissertation, The University of Michigan, Ann Arbor, 1979.

[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.