Autom Softw Eng (2009) 16: 235–261
DOI 10.1007/s10515-009-0047-y

Trading-off security and performance in barrier slicing for remote software entrusting

Mariano Ceccato · Mila Dalla Preda · Jasvir Nagra · Christian Collberg · Paolo Tonella

Received: 27 January 2009 / Accepted: 30 January 2009 / Published online: 19 February 2009

Abstract Network applications often require that a trust relationship is established between a trusted host (e.g., the server) and an untrusted host (e.g., the client). The remote entrusting problem is the problem of ensuring the trusted host that whenever a request from an untrusted host is served, the requester is in a genuine state, unaffected by malicious modifications or attacks.

Barrier slicing helps solve the remote entrusting problem. The computation of the sensitive client state is sliced and moved to the server, where it is not possible to tamper with it. However, this solution might involve unacceptable computation and communication costs for the server, especially when the slice to be moved is large. In this paper, we investigate the trade-off between security loss and performance overhead associated with moving only a portion of the barrier slice to the server and we show that this trade-off can be reduced to a multi-objective optimization problem. We describe how to make decisions in practice with reference to a case study, for which we show how to choose among the alternative options.

Keywords Program slicing · Security · Source code transformation

1 Introduction

The *Remote entrusting* problem is a particular instance of the *software integrity* problem. In software integrity, the problem is to ensure that a given program is executed

M. Ceccato (✉) · P. Tonella
Fondazione Bruno Kessler, Trento, Italy
e-mail: ceccato@fbk.eu

M.D. Preda · J. Nagra
University of Trento, Trento, Italy

C. Collberg
University of Arizona, Tucson, USA

unmodified—verifying in this way that the program has not been tampered with. In remote entrusting, 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 entrusting is that the trusting party (server) has no control over the untrusted party (client). The server cannot 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 cannot be considered known because the client user could lie about it. The client user cannot 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 entrusting 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.

It is possible to observe 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. A possible solution is to use program slicing to identify the remaining portion of the client that cannot 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 (Ceccato et al. 2007). A similar approach was used by Zhang and Gupta in order to prevent software piracy (Zhang and Gupta 2003). 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. In the remote entrusting scenario, in order to reduce the size of the computation to be moved to the server, it is possible to take advantage of the client's sub-state secured through assertions. The values of the variables secured in this way can act as *barriers* and slice computation based on 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 (Ceccato et al. 2007). If, on the one hand, moving the barrier slice is the optimal solution with regards to security, on the other hand it imposes a performance overhead. In order to limit the performance overhead, it is possible to move only a portion of the barrier slice to the server. In this work, we are interested in investigating the trade off between security loss and performance overhead when we move a portion of the barrier slice. In particular, we aim at optimizing simultaneously two conflicting objectives: minimum security loss and minimum performance overhead, where, in turn, performance overhead consists of two factors, i.e., computation cost and communication cost. The result of this multi-objective optimization problem is the fragment of barrier slice to be moved from the client to the server, which provides an optimal trade-off between security and overhead. We describe how such a trade off can be assessed in practice and how decisions can be made using an Internet game as our case study.

In Sect. 2 the remote entrusting problem is formally defined. After describing the existing solutions in Sect. 3, we present the usage of barrier slicing in Sect. 4, together with a methodology for tuning performance costs and security loss in our protection scheme (Sect. 5). Our method is then applied to a case study in Sect. 6, where results are reported and discussed. Conclusions and future works close the paper in Sect. 7.

2 Problem definition

Remote entrusting focuses on *network applications*, i.e., applications that need to access services provided by other machines over the network. Thus, the remote entrusting 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 entrusting 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 entrusting scenario

The remote entrusting scenario is shown in Fig. 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:

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

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:

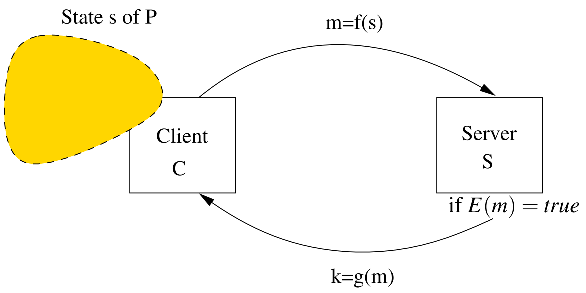

Fig. 1 Overview of remote entrusting

- 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 $A(s) = \mathit{true}$, 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 $E(m) = \mathit{true}$, where E is an assertion.

Thus, the *remote entrusting 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:

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

We can observe that a server can trivially avoid this situation by refusing to trust any client, i.e., $E(m) = \mathit{false}$ 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:

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

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 (Defense 1985) and Lampson et al. (1992). 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 (Sailer et al. 2004) and Microsoft's Palladium (Carroll et al. 2002) 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 (Seshadri et al. 2004) and Pioneer (Seshadri et al. 2005) 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 (van Oorschot et al. 2005) 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 (2003) 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. Umesh Shankar and Tygar (2004) 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.

The barrier slicing solution proposed in (Ceccato et al. 2007) uses a completely different approach, in fact it does not rely on any hardware or any precise time computation, which is hard to achieve in the presence of non-collaborative users. The idea here is 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 for remote software entrusting

A (backward) slice (Weiser 1979) 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 (Krinke 2003, 2004) is a slice computed on 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 $Cr \subseteq N$ and with barrier $B \subseteq N$ can be computed as:

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

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

When programmers use features that are difficult to fully resolve statically (such as dynamic invocations and pointers), slices may become substantially bigger, due to the conservative results of static analysis. Moreover, for practical reasons, available slicer implementations make unsound approximations (for example on libraries and on global alias analysis). This represents a well-known limitation of program slicing. However, previous studies (Ceccato et al. 2007; Krinke 2003, 2004) report that barrier slices are sensibly smaller than traditional backward slices, which reduces the conservativity issue for our protection technique.

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 $s : \mathit{Var} \to \mathit{Values}$ that associates a value with each variable in P. Given a subset $X \subseteq \mathit{Var}$ of variables, let $s_{|X}$ denote the restriction of state s on X, i.e., $s_{|X} : X \to \mathit{Values}$ 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 $\mathit{Safe} \subseteq \mathit{Var}$ be the subset of program variables that determines the usability of message k, and let $\mathit{Unsafe} = \mathit{Var} \setminus \mathit{Safe}$. This means that $s_{|\mathit{Safe}} : \mathit{Safe} \to \mathit{Values}$ 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: $A(s) = A_{\mathit{Safe}}(s_{|\mathit{Safe}}) \wedge A_{\mathit{Unsafe}}(s_{|\mathit{Unsafe}})$.

Definition 3 Let k be a server response message based on a valid client state s, i.e., $k = g(f(s))$ with $A(s) = \mathit{true}$. The *usability of* k by client in a (possibly different) state $\hat{s}$ upon execution of the communication act $S \xrightarrow{g(f(s))} C[\hat{s}]$ means:

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

We have two possible cases (see Fig. 2):

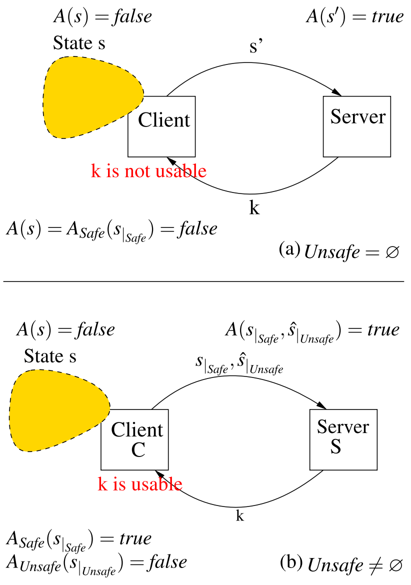
$A(s) = \mathit{false}$
$A(s') = \mathit{true}$
State s
s'
Client
Server
k is not usable
k
$A(s) = A_{\mathit{Safe}}(s_{|\mathit{Safe}}) = \mathit{false}$
(a) $\mathit{Unsafe} = \varnothing$
$A(s) = \mathit{false}$
$A(s_{|\mathit{Safe}}, \hat{s}_{|\mathit{Unsafe}}) = \mathit{true}$
State s
$s_{|\mathit{Safe}}, \hat{s}_{|\mathit{Unsafe}}$
Client C
Server S
k is usable
k
$A_{\mathit{Safe}}(s_{|\mathit{Safe}}) = \mathit{true}$
$A_{\mathit{Unsafe}}(s_{|\mathit{Unsafe}}) = \mathit{false}$
(b) $\mathit{Unsafe} \neq \varnothing$

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

1. *Unsafe* $= \varnothing$. In this case the remote entrusting problem can be solved by choosing $m = s$ and $E = A$ (i.e., the client sends the whole state). In fact, this ensures that whenever the current state $\hat{s}$ of the client C is not valid, i.e., $A(\hat{s}) = \mathit{false}$, even if the attacker sends a valid state s to the server, i.e., $A(s) = \mathit{true}$, 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, i.e., $A(s) = \mathit{true}$, to the server while the current state is $\hat{s} = (s_{|\mathit{Safe}}, \hat{s}_{|\mathit{Unsafe}})$ such that $A_{\mathit{Safe}}(s_{|\mathit{Safe}}) = \mathit{true}$ while $A(\hat{s}) = \mathit{false}$. The service $g(f(s))$ is usable, since $A_{\mathit{Safe}}(s_{|\mathit{Safe}}) = \mathit{true}$, 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 $s_{|\mathit{Safe}}$ and $s_{|\mathit{Unsafe}}$ upon each communication act.

In the present work, we do not address the problem of classifying variables into *Safe* and *Unsafe*. This classification is assumed to be carried out manually, by considering if changes of security sensitive variables can be either revealed by assertions evaluated on the server or if such changes can make the client application misfunction. In such a case, variables are regarded as intrinsically *Safe*. Otherwise, they belong to the *Unsafe* subset. 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 the solution proposed in (Ceccato et al. 2007) is to move to the server the portion (slice) of application P that maintains the variables in *Unsafe*. This ensures that attackers cannot access the variables in *Unsafe* and therefore it prevents attackers from tampering with them. Barrier slicing is used instead of regular slicing in order to limit the portion of code that needs to be moved to the server. It turns out that moving the sliced code to the server requires some extra communication between the server and the client, e.g., whenever the client requires the value of a variable in *Unsafe* the server must provide it. Moreover, the server must instantiate an execution of the moved barrier slice for each client being served, with an associated computation overhead. In the next section, we consider in detail these overheads and propose a method to trade them off with security.

Let us identify each communication act by a number $n \in \mathbb{N}$ and correspondingly partition the variables into Safe^n and Unsafe^n. $\mathbf{send}^n$ and $\mathbf{receive}^n$ 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 $\mathbf{send}^n$ 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 of a communication 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 Fig. 3. The h-th communication act precedes the n-th one. At the n-th

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

```
1   x = x * a; [barrier]
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; [criterion]
11    x = x + a;}
12  else { a = x * x; [criterion]
13    x = x + 2*a; }
14  x = 2*a;
    send^n (m_n = x);
    receive^n (k_n);
----------------------------------
    Safe^n = {x}, Unsafe^n = {a}
----------------------------------
```

communication act, we need to protect the backward slice of the definitions of variables in *Unsafe*^{n} that might reach $\mathbf{send}^n(\mathtt{m}_n)$ (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 $\mathbf{send}^h(\mathtt{m}_h)$, for example, statement 1 in Fig. 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 :

$$B_n \overset{\text{def}}{=} \{\text{reach-def}(\mathtt{x}, \mathbf{send}^h) \mid \mathtt{x} \in \mathit{Safe}^h, h \preceq n\}$$

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 $\mathbf{send}^h$ is a postdominator of reach-def(x, $\mathbf{send}^h$), 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 Fig. 3, the barrier contains just statement 1.

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

$$Cr_n \overset{\text{def}}{=} \{\text{reach-def}(\mathtt{a}, \mathbf{send}^n) \mid \mathtt{a} \in \mathit{Unsafe}^n\}.$$

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. Thus, in order to completely protect the variables is *Unsafe*^{n} we must move to the server the portion of code given by $\mathit{slice}_\sharp(Cr_n, B_n) = \{12, 10, 9, 8, 7, 6, 5, 4, 3, 2\}$.

In case it is not possible to decide which communication act precedes the current one, the union of candidate preceding communication acts has to be used as barriers.

The proposed code manipulation is supposed to occur as an automatic transformation step just before deploying the application. Code maintenance and evolution is expected to be carried out on the unchanged application, that can be later re-split and re-deployed.

5 Security vs. performance trade-off

If, on the one hand, by moving appropriate fragments of code, computed through barrier slicing, we prevent attackers from tampering with the variables in *Unsafe*, on the other hand this protection mechanism implies a performance overhead in terms of:

1. the amount of computation that needs to be carried out by the server; and,
2. the number of extra-communication acts.

In Sect. 4 we have shown that by moving the barrier slice $\mathit{slice}_\sharp(Cr_n, B_n)$ of an application P to the server we obtain complete protection for the variables in *Unsafe*^{n} with respect to the n-th communication act of P, while the performance overhead is proportional to the dimension of the barrier slice and to the number of extra communication acts. On the other hand, by leaving all the application on the client we have no protection for the variables in *Unsafe* with respect to any communication act, while the performance overhead is null. In the following we propose a methodology for tuning security loss and performance overhead when only a portion of $\mathit{slice}_\sharp(Cr_n, B_n)$ is moved from the client to the server. The idea is to measure both the performance overhead and the security that is possible to achieve by moving only subsets of $\mathit{slice}_\sharp(Cr_n, B_n)$. This measurement will provide a criterion for choosing which portions of $\mathit{slice}_\sharp(Cr_n, B_n)$ to move when there is a limit on the cost that we are willing to pay for protection.

5.1 Measure of performance overhead

Given a fragment C of code, let $\mathrm{CFG}^D(C)$ be the graph obtained by augmenting the Control Flow Graph (CFG) of C with data dependency edges, namely $\mathrm{CFG}^D = (N, E_c \cup E_d)$ where: N is the set of nodes corresponding to the instructions in C; E_c is the set of control flow edges between instructions in C and E_d denotes the def-use dependencies between instructions in C.

Figure 4 represents the control flow graph augmented with the data dependencies of the barrier slice that ensures security for variables in *Unsafe*^{n} with respect to the n-th communication act of the application in Fig. 3, i.e., $\mathrm{CFG}^D(\mathit{slice}_\sharp(Cr_n, B_n)) =$

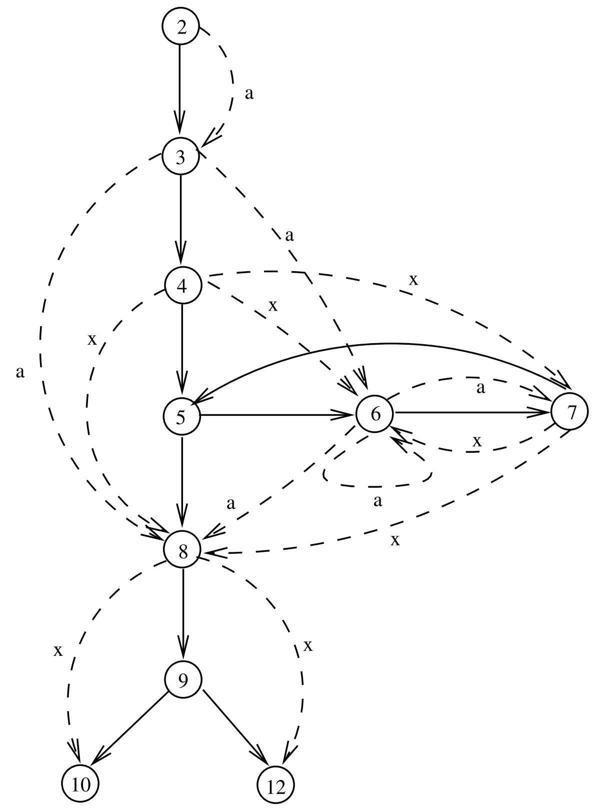
Fig. 4 CFG$^D(slice_\sharp(Cr_n, B_n))$

$(N, E_c \cup E_d)$. Here solid arrows denote the edges in E_c and dashed arrows denote the edges in E_d. Each def-use edge is labeled with the variable that causes the dependency.

The performance cost for moving a subset $S \subseteq N$ of instructions belonging to the barrier slice $slice_\sharp(Cr_n, B_n)$ is given by $(Comp(S), Comm(S))$, where $Comp(S)$ measures the amount of computation that has been moved to the server and $Comm(S)$ measures the number of extra-communication acts. Precise information about $Comp(S)$ and $Comm(S)$ is hard to obtain. We can approximate it as follows:

– We use profile information, collected by running a set of typical execution scenarios, in order to associate each node $n_i \in N$ with a value $cost(n_i)$ that represents how many times the instruction at that node is executed on average;
– We associate each control edge $\langle n_s, n_e\rangle \in E_c$ with the value $cost(\langle n_s, n_e\rangle)$ that represents how many times that edge is traversed on average (again, based on profile data);

Given $S \subseteq N$, let $Cut(S)$ be the control flow edges that connect a vertex in S to a vertex in $N \setminus S$ or vice-versa:

$$Cut(S) = \{\langle u, v\rangle \in E_c \mid ((u \in N \setminus S) \wedge v \in S) \vee (u \in S \wedge (v \in N \setminus S))\}.$$

Then we have that:

- The amount of computation carried out by the server is given by the sum of the computational cost of every instruction moved on the server:

$$Comp(S) = \sum_{n_i \in S} cost(n_i).$$

- The number of extra communication acts is given by the sum of the costs of every control edge in $Cut(S)$:

$$Comm(S) = \sum_{\langle u,v\rangle \in Cut(S)} cost(\langle u, v\rangle).$$

In fact, every edge in $Cut(S)$ implies an extra communication between the server and the client. In particular:

- for every $\langle u, v\rangle \in E_c$ where $u \in N \setminus S$ and $v \in S$, we have that after the execution of the instruction at node u the client has to ask the server to resume execution from instruction v;
- for every $\langle u, v\rangle \in E_c$ where $u \in S$ and $v \in N \setminus S$, we have that after execution of the instruction at node u the server and the client need to synchronize with each other.

For every data dependency $\langle u, v\rangle \in E_d$ on variable x where $u \in N \setminus S$ and $v \in S$, the value of x computed by the client's instruction at node u needs to be sent to the server before it executes the instruction at node v. When the client asks the server to remotely execute a set of consecutive instructions $X \subseteq S$ which includes v, the values of the variables, including x, that have been modified by the client and whose updated values are needed by the server for correct execution of X are transmitted when the control transfer occurs, i.e., when the instruction `remote-exe(L,vars)` is executed, with `L` a label that uniquely identifies the code fragment X and `vars` the following set of variables:

$$\{x \mid u \overset{x}{\rightarrow} v \in E_d \wedge u \in N \setminus S \wedge v \in X\}.$$

On the other hand, for every data dependency $\langle u, v\rangle \in E_d$ on variable x where $u \in S$ and $v \in N \setminus S$, the value of x computed by the server's instruction at node u has to be sent to the client before the execution of the instruction at node v. In particular, when the server and the client synchronize their execution through the instruction `sync(L,vars)`, the server sends the client the set `vars`, formally defined as:

$$\{x \mid u \overset{x}{\rightarrow} v \wedge u \in X \wedge v \in N \setminus S\}.$$

The proposed metrics represent an approximation of the actual computation and communication cost. We give an equal weight to all different statements and communication acts that could require, in principle, different time (e.g., complex arithmetic computation versus simple and faster statements). A more accurate estimation could be done, for example, by profiling the application under analysis to log the actual time spent on each statement and to measure the size of each message.

5.2 Measure of security loss

Given a fragment S of the barrier slice, we measure its performance overhead in terms of $\mathit{Comm}(S)$ and $\mathit{Comp}(S)$. Now we need to provide a measure for the loss in security that is implied by moving a fragment S of the barrier slice, instead of the entire slice. In the following, we assume the loss of security to be proportional to the size of the barrier slice that is kept on the client, i.e., $|N \setminus S|$. The idea is that leaving a high number of vulnerable instructions on the client increases the possibility that an attacker succeeds. However, this might not be true in general. In fact, there might be some instructions (or combination of instructions) that are more vulnerable than other ones because, when they are executed on the client, they reveal "important" information to the attacker. In this case, we could assign a weight $W(n)$ to each node in CFG$(\mathit{slice}_\sharp(Cr_n, B_n))$, such that $W(n) > W(m)$ whenever leaving instruction at node n on the client is more dangerous than leaving instruction at node m. In this case the security loss of moving a fragment S of the barrier slice is given by:

$$\mathit{SecLoss}(S) = \sum_{n \in N \setminus S} W(n)$$

In fact, by moving all the barrier slice to the server we have that $N \setminus S = \varnothing$ and the security loss is null, while by leaving all the computation on the client we have that $N \setminus S = N$ and the security loss is maximal. Clearly, weights depend on the particular application that we are considering and it might not be always possible (or easy) to assign such values to instructions. Moreover, we could think to assign a weight to combinations of instructions and not only to the single instructions.

In the following, we consider the simple case where all the nodes have the same weight and therefore the security loss is simply given by $|N \setminus S|$.

Example

Let us see how the proposed transformation and metrics work with two examples. Let $S_1 = \{5, 6, 7, 8\}$ and $S_2 = \{2, 3, 8\}$ be two subsets of the barrier slice $\mathit{slice}_\sharp(Cr_n, B_n)$ of the previous example in Fig. 4. Figure 5 shows the result of moving the subset S_1 of instructions from the client to the server. In this case the execution of instructions 5, 6, 7 and 8 is replaced on the client by the two instructions C5: `remote-exe(L1,x,a)` and C6: `sync(L1,x,a)`. In particular, C5 asks the server to execute the fragment of code at label L_1 and since the instructions in this fragment use the values of variables `x` and `a`, computed by the client instructions C3 and C4, these values are sent from the client to the server. Moreover, instruction C6 on the client, together with instruction S5: `sync(L1,x,a)` on the server, performs the synchronization of the client and the server after the execution of the fragment at label L_1 on the server. In particular, since variables `x` and `a` are used by the client in successive instructions, i.e., C8, C9, C10, C11 and C12, the server sends the updated values of these variables to the client.

A similar process is followed when moving the subset S_2 of instructions from the client to the server, see Fig. 6.

```
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    a = a + x;
      send^h(m_h);
      receive^h(k_h);
C3    a = x + a;
C4    x = x + 1;


C5    remote-exe(L_1,x,a)
C6    sync(L_1,x,a);


C7    if (c)
      then {
C8      a = 2 * x;
C9      x = x + a; }
      else {
C10     a = x * x;
C11     x = x + 2a; }
C12   x = 2a;
      send^n(m_n);
      receive^n(k_n);
```

(b)

```
      receive^h(m_h);
      x = m ;
      if A(x, a)
      then send^h(k_h);
      else exit();


S1    L_1: while (c) {
S2      a = a + x;
S3      x = x + a;}
S4      x = x * a;
S5    sync(L_1,x,a);


      receive^n(m_n);
      x = m ;
      if A(x, a)
      then send^n(k_n);
      else exit();
```

(c)

Fig. 5 Moving $S_1 = \{5, 6, 7, 8\}$: (**a**) original client, (**b**) modified client and (**c**) corresponding server

Specifically, we can observe that during the first synchronization between the server and the client there are no variables that need to be sent from the server to

```
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    remote-exe(L_1,x,a)
C3    sync(L_1)


      send^h (m_h);
      receive^h (k_h);


C4    remote-exe(L_2)
C5    sync(L_2,a)


C6    x = x + 1;
C7    while (c) {
C8       a = a + x;
C9       x = x + a; }


C10   remote-exe(L_3,x,a)
C11   sync(L_3,x)


C12   if (c)
C13   then { a = 2 * x;
C14      x = x + a; }
C15   else { a = x * x;
C16      x = x + 2a; }
C17   x = 2a;
      send^n (m_n);
      receive^n (k_n);
```

(b)

```
S1    L_1: a = a + x;
S2    sync(L_1)


      receive^h (m_h);
      x = m ;
      if A(x, a)
      then send^h (k_h);
      else exit();


S3    L_2: a = x + a;
S4    sync(L_2,a);


S5    L_3: x = x * a;
S6    sync(L_3,x)


      receive^n (m_n);
      x = m ;
      if A(x, a)
      then send^n (k_n);
      else exit();
```

(c)

Fig. 6 Moving $S_2 = \{2, 3, 8\}$: (**a**) original client, (**b**) modified client and (**c**) corresponding server

the client, because the new values of the variables modified by the server do not reach any instruction executed on the client.

In order to measure the performance overhead in the two previous examples we need the computation costs for the instructions and the communication costs for the control flow edges in the cut. Let us assume the following costs:

- $cost(2) = cost(3) = cost(4) = cost(8) = cost(9) = cost(10) = cost(12) = 1$, while $cost(6) = cost(7) = 9$ and $cost(5) = 10$;
- cost of the control flow edges in E_c: $cost(\langle 2,3\rangle) = cost(\langle 3,4\rangle) = cost(\langle 4,5\rangle) = cost(\langle 5,8\rangle) = cost(\langle 8,9\rangle) = cost(\langle 9,10\rangle) = 1$, while $cost(\langle 5,6\rangle) = cost(\langle 6,7\rangle) = cost(\langle 7,5\rangle) = 9$;
- security weight of nodes is $W(n) = 1$ for each node $n \in N$.

We have that $Comp(S_1) = cost(5) + cost(6) + cost(7) + cost(8) = 10 + 9 + 9 + 1 = 29$ and $Comm(S_1) = cost(\langle 4,5\rangle) + cost(\langle 8,9\rangle) = 2$ and $SecLoss(S_1) = W(2) + W(3) + W(4) + W(9) + W(10) + W(12) = 1 + 1 + 1 + 1 + 1 + 1 = 6$.

While $Comp(S_2) = cost(2) + cost(3) + cost(8) = 3$ and $Comm(S_2) = cost(\langle 3,4\rangle) + cost(\langle 5,8\rangle) + cost(\langle 8,9\rangle) = 3$ and $SecLoss(S_2) = W(4) + W(5) + W(6) + W(7) + W(9) + W(10) + W(10) = 1 + 1 + 1 + 1 + 1 + 1 + 1 = 7$.

Therefore, $(Comp(S_1), Comm(S_1), SecLoss(S_1)) = (29, 2, 6)$, while $(Comp(S_2), Comm(S_2), SecLoss(S_2)) = (3, 3, 7)$.

While communication and computation costs could be in principle measured, for example through profiling, security loss is different, in that it is more difficult to measure, being more subjective. In fact, different considerations about security could lead to different solutions (as described below).

Leaving the whole barrier slice on the server would represent a quite secure solution, because (according to our attack model) no tampering can be applied in this case (as in Ceccato et al. 2007) to the security sensitive code. By moving back some security sensitive code to the client, we accept a trade off that is less secure (i.e., not sound), in order to achieve better performance.

5.3 Trade-off

For every $S \subseteq N$, where N is the set of nodes of CFG($slice_\sharp(Cr_n, B_n)$), the performance overhead $(Comp(S), Comm(S))$ and the security loss $SecLoss(S)$ can be represented as a point in a three dimensional space where the axes measure the computation cost, the communication cost and the security loss. Choosing a trade-off between performance overhead and security loss means choosing among the points associated with each S in this three dimensional space. One way to make such a selection is by means of multi-objective optimization and Pareto optimality.

Pareto optimality (Collette and Siarry 2004) was originally defined in economics, but has since been applied to several areas including software engineering. A *multi-objective optimization problem* consists of finding a set of solutions which optimize (minimize) a set of cost functions (also known as objective functions). A solution x to the given optimization problem is said to *dominate* another solution y $(x \succ y)$ *iff* all objective function values for x are greater than or equal to the corresponding objective function values computed for y. All solutions that are not dominated by any other solutions are said to belong to the *Pareto optimal solution*, the *Pareto front* being the corresponding objective function values.

Let us consider the costs that characterize the trade-off between performance overhead and security loss: $(Comp(S), Comm(S), SecLoss(S))$. They define a multi-objective optimization (cost minimization) problem, that can be addressed by the heuristic algorithms available for this category of problems (see next section). The

Pareto optimal solution for this problem is a set of triples of costs that are not dominated by any other triple associated with a barrier slice subset S.

For the examples in Figs. 5, 6 the respective cost triples are: (29, 2, 6), (3, 3, 7). Correspondingly, the solution S_1 is not dominated by S_2 and vice-versa S_2 is not dominated by S_1. In other words, S_1 and S_2 are *incomparable* solutions to the multi optimization problems. If they are both not dominated by any other solution, they belong to the Pareto front and are reported as possible alternative solutions to the user. The final choice is based on a manual, subjective assessment of the trade off among the solutions in the Pareto front. In our example, assuming S_1 and S_2 are in the Pareto front, the user may judge a computation cost of 29 unacceptable and opt for the second solution.

Once all the data points are available, the trade off can be done on the graph that shows the Pareto front. Since such a graph is 3-dimensional in our case, it may be more convenient to project it onto two dimensions and give a parametric plot of the third one. Figure 7 shows the projected Pareto front for our example, with the x-axis and y-axis measuring respectively *Comp* and *Comm*. Data points that have the same value of *SecLoss* are connected with a line and drawn in the same color. *SecLoss* values are reported in a label near the line edge. For completeness, also S_1 and S_2 are reported on the graph, even if, considering all the data point, they have been identified as not belonging to the Pareto front (i.e., they are dominated by other cuts).

In Fig. 7 an area should be identified that satisfies the overhead requirements, let's say *Comp* < 5 and *Comm* < 5. In this area the solution is selected that provides the smallest value for *SecLoss* (i.e., 5). Then, depending on subjective evaluations, the solution might be slightly changed. For example, a bigger penalty in security could be preferred if the gain in performance is substantial. For instance, by choosing the point with *SecLoss* $= 6$, both *Comp* and *Comm* improve by 1.

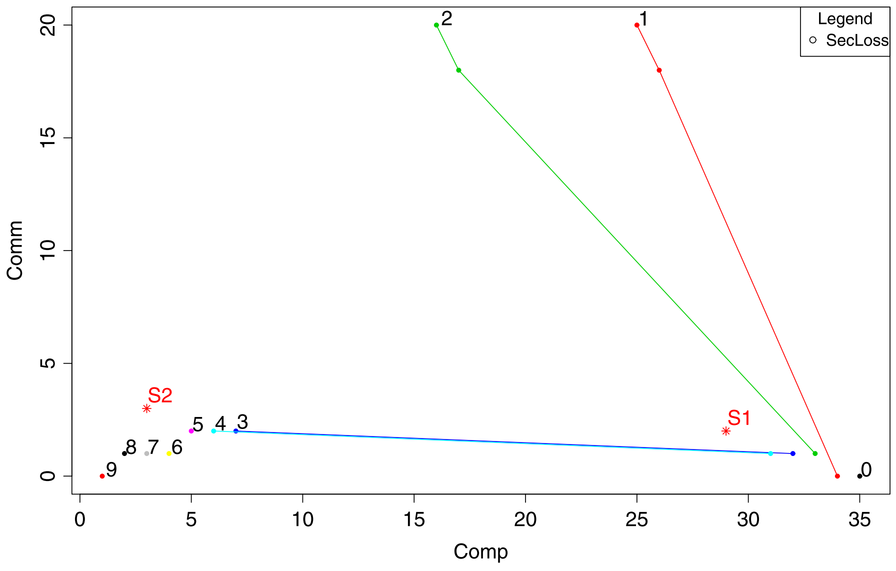

Fig. 7 *Comp* vs *Comm* for the example

5.4 Search-based computation of optimal cut

Our goal is optimizing three objective functions at the same time, namely, computation cost, communication cost and security loss cost. The search space of the barrier slice subsets (cuts) optimizing such three objective functions is huge, consisting of all subsets of the CFG nodes in the barrier slice: $\wp(N)$, with N the set of nodes in the barrier slice. The result we aim at is a set of CFG cuts that belong to the Pareto optimal solution. No algorithm providing the exact solution is available for this problem, hence we must resort to heuristics. Search-based techniques, such as hill-climbing (Tovey 1985), simulated annealing (Kirkpatrick et al. 1983), genetic algorithms (Goldberg 1989) and ant colony optimization (Dorigo and Gambardella 1997) can be employed for such a purpose. In this work we use a relatively simple heuristic, hill-climbing, since it already exhibits good performance. In our future work we will compare its performance with that of other alternative search-based heuristics.

Figure 8 shows the pseudo-code of the hill climbing algorithm tailored to the problem under consideration. Initially, a random set of candidate cuts is generated (lines 2–6), under the control of the parameter *InitSize*, which determines its size. The hill

Fig. 8 Hill climbing algorithm to compute Pareto-optimal cuts

```
Procedure: hillClimbing
Input:
   CFG = <N, E>: barrier slice CFG
   F1: P(N) -> Real: computation cost
   F2: P(N) -> Real: communication cost
   F3: P(N) -> Real: security loss cost
Output:
   CUTS: set of CFG cuts that minimize F1, F2, F3
Parameters:
   InitSize: initial size of CUTS
Begin
1     CUTS := EmptySet
2     NewCUTS := EmptySet
3     While size(NewCUTS) < InitSize
4         NewCUTS := add(NewCUTS, randomSubset(N))
5     End while
6     NewCUTS := removeDominated(NewCUTS)
7     While CUTS <> NewCUTS
8         CUTS := NewCUTS
9         For EachCut in CUTS, NewCut in neighbors(EachCut)
10            isNewCutDominated := false
11            For CurrCut in CUTS
12                If F1(CurrCut) < F1(NewCut) and
13                   F2(CurrCut) < F2(NewCut) and
14                   F3(CurrCut) < F3(NewCut)
15                      isNewCutDominated := true
16                End if
17            End for
18            If not isNewCutDominated
19               NewCUTS := add(NewCUTS, NewCut)
20            End if
21        End for
22        NewCUTS := removeDominated(NewCUTS)
23    End while
24    Return CUTS
End
Procedure: removeDominated
Input: Cuts: set of cuts
Output: NewCuts: subset of non dominated cuts
Begin
25    NewCuts := Cuts
26    For cut1 in Cuts, cut2 in Cuts
27        If F1(cut1) <= F1(cut2) and
28           F2(cut1) <= F2(cut2) and
29           F3(cut1) <= F3(cut2) and
30           (F1(cut1) <> F1(cut2) or
31            F2(cut1) <> F2(cut2) or
32            F3(cut1) <> F3(cut2))
33              NewCuts := remove(NewCuts, cut2)
34        End if
35    End for
36    Return NewCuts
End
```

climbing loop starts at line 7 and terminates at 23. As long as better cuts are found, the algorithm keeps improving the current heuristic solution.

At the core of the hill climbing algorithm is the computation of the neighbors of the current solution (line 9). In fact, the current solution is improved whenever one or more cuts in the neighborhood of the currently selected cuts is not dominated by any available cut. In such a case, the improving cut is added to the new solution (*NewCUTS*, line 19).

Computation of the neighbors of a cut (not shown in Fig. 8) is based on two atomic operations: (1) add node; (2) remove node. For a given cut (set of nodes), neighboring cuts are generated by adding in turn each node in the CFG not already in the cut and by removing in turn each node in the cut.

After computing the new set of candidate cuts, only non-dominated cuts are kept (line 22). The condition that specifies when a cut is dominated and must be removed from the current solution is detailed inside procedure *removeDominated*, between line 27 and 32.

The output of the algorithm consists of a set of solutions (cuts) that are not dominated by their neighbors (local optima). They are reported to the user for final, manual selection of the actual solution to adopt.

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 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 Experimental data

Variables in *Unsafe* are used as slicing criteria, whereas variables in *Safe* are used as barriers for the computation of the barrier slice. Statements in the barrier slice

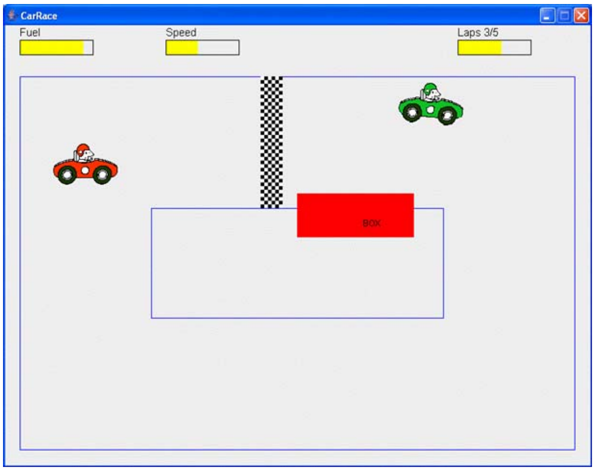

Fig. 9 Screen shot of CarRace network game

are instrumented to trace the application run and determine the computation costs empirically.

A typical execution of the application (i.e., an entire race) has been traced in order to obtain typical data about how many times each statement is executed and how many times each control dependence is traversed. These values represent edge and node weights in the CFG. Because no information was available about the security impact of the different statements in the slice, the assumption of uniform weights was made. All the statements have been assigned a security loss cost equal to one.

The three trade-off costs (computation, communication and security loss) have been subjected to multi objective optimization using the hill-climbing search algorithm with an initial set of 1,000 cuts. Since this algorithm is non-deterministic, it was run multiple times (i.e., 100 times) to reduce the probability of converging to local optima. The results coming from all the runs have been finally merged and the final Pareto front has been computed. In Fig. 10, each data point represents a cut in the Pareto front. x and y axes measure, respectively, the computation and communication costs (*Comp* and *Comm*) while the labels report *SecLoss*. Data with the same *SecLoss* value are connected with a line and they have the same color.

The barrier slicing solution of our previous paper (Ceccato et al. 2007) corresponds to the point at the bottom-right of the diagram, with the maximum *Comp* and the minimum (zero) *SecLoss*. This cut corresponds to moving all the statements of the barrier slice to the server. Going from left to right, *Comp* increases while the *SecLoss* decreases because we encounter cuts with more and more statements moved to the server. Of course, solutions with the same number of statements left on the client (the same *SecLoss*) could have different computational and communication overhead (*Comp* and *Comm*), depending on which statements are going to be moved to the server.

At the two extremes, when either the client or the server contain few statements, the communication overhead is small. On the other hand, for intermediate values of *Comp*, there is quite a big communication cost. This can be explained by the presence of quite well isolated parts of the application that have not so many dependencies with

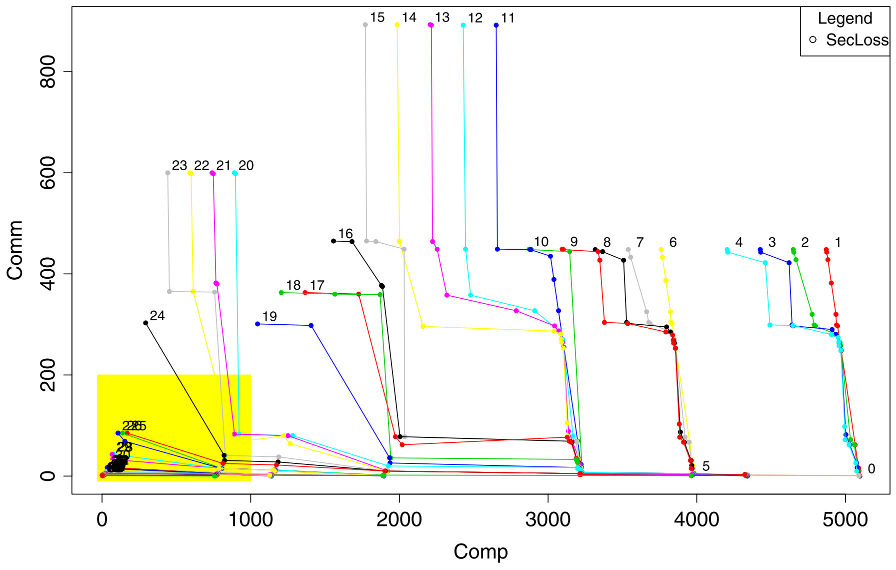

Fig. 10 *Comp* vs *Comm* for CarRace

each other. When they are entirely on the same host they do not need to traverse the cut often (low communication cost), but when they are split, they cause an increase in the *Comm* cost, corresponding to intermediate values of *Comp*.

6.3 Trade-off on the graph

In Fig. 10, all points in the graph represent a possible solution to the trade-off problem. Which one to select depends on the most important costs to minimize. CarRace is an on-line game to be used interactively, so a big computation and communication overhead is unacceptable, because it would cause a noticeable delay in the game. Moreover, for home users the communication overhead should not exceed the bandwidth of a typical domestic connection. Let us assume that these two constrains correspond to $Comp < 1000$ and $Comm < 200$. This means that we can restrict the search in the portion of the graph corresponding to the highlighted (bottom-left) area in Fig. 10. For convenience, a zoom-in of this area is shown in Fig. 11.

Among all the data points in Fig. 11, the most secure solution is represented by P_{20}, the point with $SecLoss = 20$. However, if we are willing to accept a slightly more insecure solution we could consider also the point P_{25}. In this case the lower level of security (25% increase in *SecLoss*) would be paid back by relevant improvement in the computational overhead (−82% in *Comp*) and a negligible communication penalty (+2% in *Comm*). Details about the costs of the two candidate solutions can be found in Table 1.

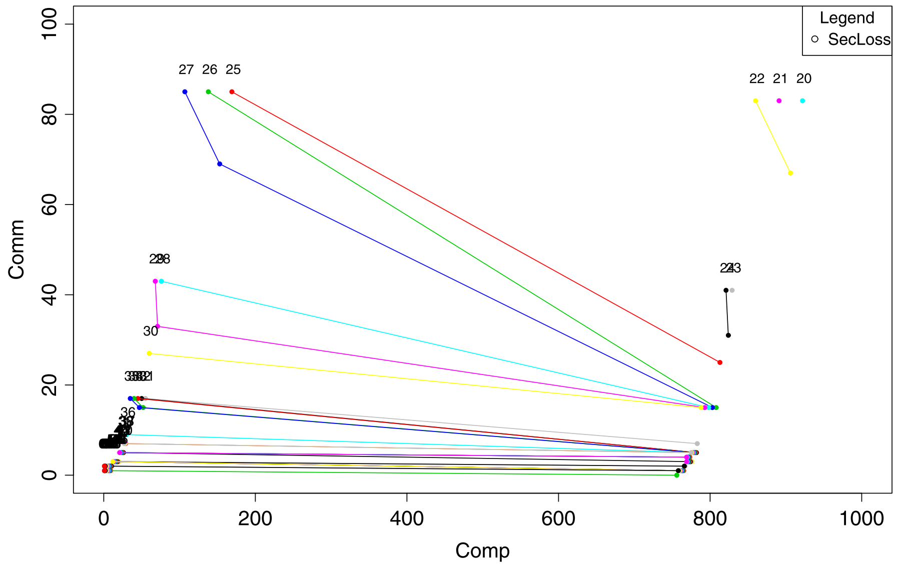

Fig. 11 Detail of *Comp* vs *Comm* for CarRace

$$\begin{tikzcd}
\text{exec} & & & \\
\text{changeSpeed} & & & \\
& & \text{updateGas} \arrow[r] & \text{isInBox} \\
\text{run} \arrow[r] & \text{update} \arrow[ur] \arrow[dr] & & \\
& & \text{updateLaps} \arrow[r] \arrow[dr] & \text{isInStartLine} \\
& & & \text{isInCheckLine}
\end{tikzcd}$$

Fig. 12 Call graph of the methods involved in the two candidate solutions P_{20} and P_{25}

6.4 Trade-off on the code

By just looking at the graph, both P_{20} and P_{25} seem good candidate solutions. In order to decide which one to adopt, a deeper investigation could be done by browsing the involved segments of code.

Figure 12 shows the call graph for the portion of code involved in P_{25} and P_{20}. While the methods *changeSpeed* and *exec* are isolated, the rest of the call graph is very connected. The *run* method belongs to a thread that periodically calls *update*, which delegates to *updateGas* and *updateLaps*. Then, the call chain ends at *isInStartLine*, *isInCheckLine* and *isInBox*.

Table 1 Trade off among the costs of two candidate solutions

	Comp	Comm	SecLoss
P_{20}	922	83	20
P_{25}	169	85	25
	−82%	+2%	+25%

Table 2 Portion of slice left on the client for two candidate solutions

Class	Method	Line	cost	P_{20}	P_{25}
CircuitModel	isInStartLine	63	222	X	X
CircuitModel	isInCheckLine	71	160	X	X
CircuitModel	isInBox	80	222	X	X
MovingCarModel	updateLaps	150	222	X	X
MovingCarModel	updateLaps	151	222	X	X
MovingCarModel	updateLaps	152	160	X	X
MovingCarModel	updateLaps	153	124	X	X
MovingCarModel	updateLaps	154	191	X	X
MovingCarModel	updateGas	111	222	X	X
MovingCarModel	updateGas	118	214	X	X
MovingCarModel	updateGas	119	214	X	X
MovingCarModel	update	99	222	X	X
MovingCarModel	update	104	222	X	X
MovingCarModel	update	105	222	X	X
UpdateThread	run	191	222	X	X
ServerHandler	exec	44	225	X	X
ServerHandler	exec	52	222	X	X
ServerHandler	exec	53	222	X	X
ServerHandler	exec	54	222	X	X
ServerHandler	exec	55	222	X	X
MovingCarModel	changeSpeed	52	156		X
MovingCarModel	changeSpeed	53	150		X
MovingCarModel	changeSpeed	67	149		X
MovingCarModel	changeSpeed	68	149		X
MovingCarModel	changeSpeed	72	149		X

Table 2 shows what lines of the slice are left on the client either for P_{20} or P_{25}. The first three columns reports class, method name and line number. The next column contains the computation cost for each line, while the security cost is assumed to have constant value 1. In the last two columns an “X” indicates that the corresponding line is left on the client, either by P_{20} or P_{25}.

The solution associated with P_{25} contains a super-set of lines of P_{20}. The only difference is that the former contains 5 more lines from method *changeSpeed*.

The first three statements of Table 2 come from the class *CircuitModel*. They are the whole bodies of three small methods. These three methods are called mainly in the first part of method *updateLaps* of class *MovingCarModel*. The responsibility of *updateLaps* is to check whether a new lap is completed and, in that case, to update the lap variable (*Lap* is in *Unsafe*). A part of this method is in P_{25} and P_{20} (lines 150 to 154). The rest of the method (lines 155 to 164) is left on the server by both of the solutions. Method *updateLaps* is periodically executed, but while the first part of this method always runs about 200 times, the second part runs only when the car reaches the end of a sector (half lap) so it is executed more rarely (about 10 times). Thus the frequently executed part of code can be left on the client, while the rarely executed one, that actually increments the number of performed laps, should be moved to the server.

Method *updateGas* is also only partially in P_{25} and P_{20}. The responsibility of this method is to change the fuel level. Depending on whether the car is running or is refueling at the box, the fuel is reduced or increased. Since the second case is more rare, the first part of the method runs more frequently and it can be left on the client. The second part is, vice-versa, moved to the server, causing only minimal overhead, but protecting the behavior of the application against an unfair arbitrary refueling.

Methods *update* and *run* contain just calls to the former update methods, so they can safely remain on the client. The portion of the *exec* method that P_{25} and P_{20} leave on the client are executed when a message is received from the server that contains the position of the opponent, to update the screen. Even if on the client, these instructions do not represent a security problem, even in case of tampering.

Method *changeSpeed* represents the only difference between P_{20} and P_{25}. The lines of code left on the client by P_{25} are the ones that evaluate whether to increase the speed of the car when a corresponding key-pressed event is received. In fact, if the fuel level is low, the speed should not be increased. They represent a major security vulnerability of P_{25} with respect to P_{20} and would allow a quite serious attack. For this reason the solution associated with P_{20} is preferable.

In case this functionality is left on the server, an assertion could be used to check that change speed events come at a valid rate, such that the car acceleration respects the physical constrains (i.e. maximum acceleration) implemented in the game.

7 Conclusion and future work

In this paper, we address the problem of remote software entrusting, i.e., a trusted server has to verify the healthy execution of a given application on a remote client before delivering any service to it. In our previous work (Ceccato et al. 2007), we proposed a solution based on the use of barrier slicing to identify the portions of the client code that have to be moved to the server in order to protect otherwise unsafe variables. While this technique ensures complete protection (an attacker cannot tamper with what he/she cannot access), it imposes a performance overhead in terms of computation that has to be carried out by the server and of extra communication acts that are needed to synchronize the remote execution. On the other hand, when leaving all the computation on the client we have no overhead and the maximal degree of insecurity. In this work, we make a step forward and we investigate the trade-off between performance overhead and security when moving only portions of the barrier slice.

In order to handle this trade-off, we describe a method for measuring the computation and communication overhead and the security loss. Each subset of the full barrier slice is a cut in the control flow graph augmented with data dependencies and can be represented as a point in a three dimensional space, where the axes measure the computation cost, the communication cost and the security loss. Hence, the problem of finding an optimal trade-off can be turned into a Pareto optimality problem, where the three costs involved in the trade-off are to be simultaneously minimized.

We computed optimal solutions for a case study application. By inspecting the graphical representation of the Pareto front, we identified two candidate points that represent two acceptable trade-off solutions between security and overhead. Based on further security considerations, we have been able to made a decision on the final solution. This required some manual code browsing.

In our future work, we are going to further investigate how to properly measure the security loss associated to the instructions of a given program. Moreover, we plan to combine our protection scheme with other ones. For example, by combining obfuscation and barrier slicing we may be able to increase the resistance of obfuscation, based on the unavailability of portions of relevant computation, removed from the client. Moreover, the possibility to change dynamically the cut of the barrier slice (e.g., by replacing portions of the client at run time) opens to the possibility of continuously changing the sensible computation left on the client, hence adding further protection to the application. We also intend to continue our experimental investigation with more case studies, so as to understand the involved trade-off in more detail.

Acknowledgements This work was supported by funds from the European Commission (contract No. 021186-2 for the RE-TRUST project)

References

Carroll, A., Juarez, M., Polk, J., Leininger, T.: Microsoft “Palladium”: a business overview. Microsoft Content Security Business Unit, August 2002

Ceccato, M., Dalla Preda, M., Nagra, J., Collberg, C., Tonella, P.: Barrier slicing for remote software trusting. In: Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007), Paris, France, pp. 1–10 (2007)

Collette, Y., Siarry, P.: Multiobjective Optimization: Principles and Case Studies. Springer, Berlin (2004)

Defense, D.: Trusted computer security evaluation criteria. Washington, DC DOD 5200.28-STD (1985)

Dorigo, M., Gambardella, L.M.: Ant colony system: a cooperative learning approach to the traveling salesman problem. IEEE Trans. Evol. Comput. **1**(1), 53–66 (1997)

Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Kluwer Academic, Boston (1989)

Kennell, R., Jamieson, L.H.: Establishing the genuinity of remote computer systems. In: Proceedings of 12th USENIX Security Symposium (2003)

Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science **220**(4598), 671–680 (1983)

Krinke, J.: Barrier slicing and chopping. In: Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation, pp. 81–87 (2003)

Krinke, J.: Slicing, chopping, and path conditions with barriers. Softw. Qual. J. **12**(4), 339–360 (2004)

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

Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and Implementation of a TCG-based Integrity Measurement Architecture, pp. 223–238 (2004)

Seshadri, A., Perrig, A., van Doorn, L., Khosla, P.K.: SWATT: software-based attestation for embedded devices. In: IEEE Symposium on Security and Privacy, pp. 272–283 (2004)

Seshadri, A., Luk, M., Shi, E., Perrig, A., van Doorn, L., Khosla, P.K.: 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 23–26, pp. 1–16 (2005)

Tovey, C.A.: Hill climbing with multiple local optima. SIAM J. Matrix Anal. Appl. **6**(3), 384–393 (1985)

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

van Oorschot, P., Somayaji, A., Wurster, G.: Hardware-assisted circumvention of self-hashing software tamper resistance. IEEE Trans. Dependable Secure Comput. **2**(2), 82–92 (2005)

Weiser, M.D.: Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. PhD dissertation, The University of Michigan, Ann Arbor (1979)

Zhang, X., Gupta, R.: Hiding program slices for software security. In: CGO ’03: Proceedings of the International Symposium on Code Generation and Optimization, Washington, DC, USA, pp. 325–336 (2003)