Empirical assessment of the effort needed to attack programs protected with client/server code splitting

**Empirical Software Engineering (EMSE)**
https://doi.org/10.1007/s10664-019-09738-1

# Empirical Assessment of the Effort Needed to Attack Programs Protected with Client/Server Code Splitting

**Alessio Viticchié, Leonardo Regano, Cataldo Basile, Marco Torchiano, Mariano Ceccato, Paolo Tonella**

Published online: 25 July 2019

**Abstract** Context: Code hardening is meant to fight malicious tampering with sensitive code executed on client hosts. Code splitting is a hardening technique that moves selected chunks of code from client to server. Although widely adopted, the effective benefits of code splitting are not fully understood and thoroughly assessed.

Goal: The objective of this work is to compare non protected code vs. code splitting protected code, considering two levels of the chunk size parameter, in order to assess the effectiveness of the protection – in terms of both attack time and success rate – and to understand the attack strategy and process used to overcome the protection.

Method: We conducted an experiment with master students performing attack tasks on a small application hardened with different levels of protection. Students carried out their task working at the source code level.

Results: We observed a statistically significant effect of code splitting on the attack success rate that, on the average, was reduced from 89% with unprotected clear code to 52% with the most effective protection. The protection variant that moved some small-sized code chunks turned out to be more effective than the alternative moving fewer but larger chunks. Different strategies were identified yielding different success rates. Moreover we discovered that successful attacks exhibited different process w.r.t. failed ones.

Conclusions: We found empirical evidence of the effect of code splitting, assessed the relative magnitude, and evaluated the influence of the chunk size parameter. Moreover we extracted the process used to overcome such obfuscation technique.

## 1 Introduction

Software contains valuable assets that are targeted by a multitude of hackers (or better crackers) that may be interested in misusing software applications, extracting intellectual property, or use them as vehicle for broader attacks (e.g., malware infections).

Cracking can be performed at the attacker machine, where an arsenal of tools is available to reverse engineer and tamper with the target applications. These tools include static and dynamic analysis tools, debuggers, disassemblers, decompilers, etc. This scenario has been referred to as Man at the End (MatE) attack [13].

One of the basic secure software engineering guidelines prescribes to allocate security sensitive computations (e.g., access control mechanisms) to the server and to leave on the client only computations that do not expose any attack surface or that can be verified on the server side [15]. This idea has been turned into a systematic technique, called Client/Server Code Splitting [6], which aims at automatically redistributing security sensitive computations between client and server. The degree of protection can be tuned, since the code portion to be split can be larger or smaller. Correspondingly, the overhead incurred by the application is also variable. In fact, the functionalities removed from the client are replaced by messages exchanged with the server to restore the data values that are no longer computed on the client.

Alessio Viticchié, Leonardo Regano, Cataldo Basile, Marco Torchiano,
Politecnico di Torino, Torino, Italy
E-mail: {first.last}@polito.it

Mariano Ceccato
Fondazione Bruno Kessler, Trento, Italy
E-mail: {last}@fbk.eu

Paolo Tonella
Università della Svizzera Italiana (USI), Lugano, Switzerland
E-mail: {first.last}@usi.ch

While exposing less sensitive computations to the client is in principle beneficial in terms of protection, no study has assessed how such benefits translate into increased attack obstacles as the size of the split code chunk is varied. Ours is the first empirical study designed and conducted with the explicit purpose of assessing such benefits.

Prevention of any possible MatE attack by means of code obfuscation is theoretically impossible, as already showed for other hardening techniques [2]. Hence, we assess the effectiveness of Client/Server Code Splitting by evaluating also the delay it introduces when attackers perform attack tasks. In fact, an attack task can be deemed unsuccessful when the attacker is not able to achieve her/his goals within a reasonably long time frame, because a longer time investment on the attack might not be justified by the attack reward.

The experiment consisted in asking a group of master students, who are involved as subjects, to perform an attack task on a software application hardened by means of Client/Server Code Splitting. We used two protection profiles (i.e., treatments): (i) we remove from the client application a single code slice of large size (to be executed on the trusted server) or (ii) we split multiple slices, each of them of small size. Participants were divided into three groups and provided with the source of code of the client after the application of one of the treatment or the control case (unprotected code). The attack tasks consisted of modifying the behaviour of an open source application in a time frame of 2h. Moreover, students were asked to answer a pre-questionnaire to evaluate their programming abilities and a post-questionnaire to describe their attack activities.

The experiment showed a statistically significant effect of code splitting on the attack success rate. On the average, attack success rate was reduced from 89% with unprotected clear code to 52% with the most effective protection. The protection variant that moved some small-sized code chunks turned out to be more effective than the alternative moving fewer but larger chunks. Different strategies were identified yielding different success rates. Moreover, we discovered that subjects that successfully performed their attack task exhibited a different attack strategy from subjects that did not succeed.

The paper is organized as follows. Section 2 sketches the Client/Server Code Splitting protection and introduces assets and possible attacks against the target application, the object of our experiment. Section 3 presents the design of the experiment, the research questions, the experimental procedure and the analysis method. Section 4 presents the results of the analysis of the data collected with the experiment and the analysis of the attack process, reconstructed from the participants' questionnaires. Section 5 discusses how the results of the experiment can be used to improve the effectiveness of Client/Server Code Splitting. Section 6 introduces related work on the evaluation of the effectiveness of protection techniques, both theoretical and based on empirical experiments. Finally, Section 7 draws conclusions and sketches future works.

## 2 Background

Despite the proved theoretical impossibility of building general purpose obfuscators [2], implementations of obfuscators do exist and they are actually used in practice. Available obfuscators provide limited though effective protection against malicious reverse engineering, transforming a program into a functionally equivalent version which is harder to understand for an attacker even through automatic code analysis tools [12].

For software developers, one of the most difficult challenges is ensuring that a malicious user cannot tamper with their application to alter its expected behavior. Users of applications that run on client devices must be considered a primary source of threats, since they could be interested in making the program execute in a way that might give them benefits of various types. As client users, attackers have full control of the application execution environment and they have the possibility to use any sort of methods to attack the program, from running dynamic or static analysis tools to reverse-engineering the entire application. This scenario, i.e., the MatE attacks, has recently received the attention of several researchers.

*Client/Server Code Splitting* [6] is a protection technique that aim at reducing the attack surface that can be potentially targeted by attackers. This protection removes portions of the application that are considered unsafe, in that they are attackable by a malicious user, and moves them to a secure server, where they run in a safe, trusted environment. To be deployed, this technique requires the identification of the parts of the application that needs to be protected (performed for instance by means of barrier slicing) and moved to the server (e.g., by splitting them off), as well as the transformation of the original code to make the new client and the trusted server communicate for synchronized data exchange.

The rest of this section covers more details about Client/Server Code Splitting, by means of a running example, which is the object of our experiment. The purpose of this paper is to assess the effectiveness of the Client/Server Code Splitting protection, therefore, we do not to repeat all the details of the technique. Interested readers can find more information about the technique and the validations already performed in previous papers [6,5,24].

The trade off between security loss and run time performance of the program being protected was assessed experimentally on a car race game [5]. Client/Server Code Splitting has been applied to the car race game, so as to produce an initial set of 1,000 cuts (subsets of the barrier slice) of the program to be protected. Such cuts were then optimized by a search based algorithm. The resulting Pareto front provides a trade off between computational cost and communication overhead of the cut, on one hand, and corresponding security loss, on the other hand. The security level of the applied protection as well as the security loss were estimated based on the variations induced by the protection on a set of code

metrics. Therefore, the validation performed in such previous work took into account just quantitative indicators of the level of protection, without considering the actual comprehension ability of humans. This paper perfectly complements the past approach based on objective measures with the empirical assessment of the level of protection Client/Server Code Splitting can provide when humans try to break the protection.

Note that techniques and approaches that resort to trustworthy servers to perform security sensitive operations are increasing their importance and are becoming widely used in several scenarios. Indeed, nowadays, the assumption that a client application interacts with a remote server to perform its tasks is almost always valid. For instance, applications running on mobile devices, like smartphones and tablets, are commonly delegating to servers tasks that are computationally too demanding or that may affect the application assets. This is evident for games, where developers typically need to address players who cheat to gain unfair advantages and for security applications that store cryptographic material on the server-side, still performing some security sensitive operations on the client. Also web-based applications, which use obfuscation techniques to make the client-side code less comprehensible (e.g., javascript obfuscation), are adding an additional layer of server-side verification by splitting functions (e.g., repeating the same checks on client and server) to limit the risk of clients altering the intended application behaviour. Therefore, despite the differences in languages and technology, the assessment of Client/Server Code Splitting technique may allow us to understand whether a robust level of protection is generally possible with the help of automated techniques.

### 2.1 Reference Example

The mechanics of Client/Server Code Splitting are presented using a running example, namely SpaceGame[1], that is also the software system used to conduct our empirical study. SpaceGame is an open source program that demonstrates GAME (Geometrical Ascii Multi-game Environment), a C language framework for creating geometrical games with `ncurses` text screens.

In SpaceGame, a space ship is moved on the screen in an area where other objects also move. Being a demonstrator, the player has not a goal: the space ship just moves in the screen in areas that are not occupied by other objects. The spaceship is initially created in the space at a starting position. Then, its position is changed by a "pilot" function, which reads the user inputs at every key press and determines the movements of all objects on the screen. SpaceGame uses several data structures to maintain information about the objects to draw on the screen. These data structures are updated by the functions that move objects.

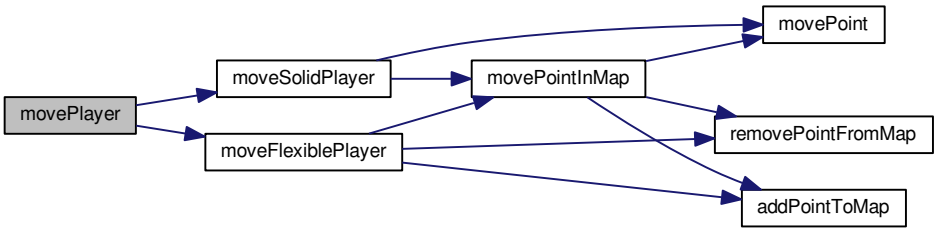



Fig. 1: Call graph below `movePlayer` in the SpaceGame program

When tampering with games, an usual attack goal is gaining an unfair advantage over other competing players. In our experiment, we simulated this scenario by defining an attack task that requires the modification of the source code so that every key press moves the space ship twice as fast as allowed by the game rules. Given this attack task, the relevant assets are functions that manipulate the data structures related to the position of the space ship (see Fig. 1 for the portion of call graph below function `movePlayer`):

- `keyboardPilot` is the pilot function that processes user inputs and determines the next move of the spaceship;
- `movePlayer` moves an object after having verified that it does not crash with other objects or cross the space boundaries. Depending on the object type, this function calls function `moveSolidPlayer` to move SOLID objects (i.e., objects that cannot pass over other objects) and function `moveFlexiblePlayer` for FLEXIBLE ones (i.e., for objects that can pass over other objects);
- `moveSolidPlayer`, since our space ship is a SOLID object, this function is called every time the spaceship is moved;
- `movePoint` is the function that computes the new position of all the points that form each object depending on the movement direction (objects may be composed of multiple points on the screen);
- `movePointInMap` updates the map, the internal structure that describes the status of each "space area" point. It first removes the information of the moving object from the current position (by calling `removePointFromMap`), then it

[1] SpaceGame can be obtained from SourceForge at `https://sourceforge.net/projects/game/`.

computes where the object will be placed (by calling `movePoint`, depending on the movement direction). Finally, it updates the map by recording the presence of the moving point in the new position (by calling `addPointToMap`);

```
1  void movePoint(Point* point, int
       direction){
2     assert (point != NULL);
3     if(direction==STOP){
4     return;
5     }
6     switch(direction) {
7     case N:
8        point->row--; break;
9     case NE:
10       point->row--;
11       point->col++; break;
12    case E:
13       point->col++; break;
14    case SE:
15       point->row++;
16       point->col++; break;
17    case S:
18       point->row++; break;
19    case SW:
20       point->row++;
21       point->col--; break;
22    case W:
23       point->col--; break;
24    case NW:
25       point->row--;
26       point->col--; break;
27 }
```

(a) Original code

```
1  void movePoint(Point* point, int
       direction){
2     assert (point != NULL);
3     if(direction==STOP){
4        return;
5     }
6     switch(direction) {
7     case N:
8        point->row-=2; break;
9     case NE:
10       point->row-=2;
11       point->col+=2; break;
12    case E:
13       point->col+=2; break;
14    case SE:
15       point->row+=2;
16       point->col+=2; break;
17    case S:
18       point->row+=2; break;
19    case SW:
20       point->row+=2;
21       point->col-=2; break;
22    case W:
23       point->col-=2; break;
24    case NW:
25       point->row-=2;
26       point->col-=2; break;
27 }
```

(b) Attacked code.

Fig. 2: Source code of the `movePoints` function before and after the attack task

Therefore, an attacker may modify the function `movePoint` by just changing the unitary increment or decrement of the position into a double increment/decrement. The code fragment on the left of Fig. 2 is the original code, while the code fragment on the right shows how all increments have been changed to accomplish the attack.

Although this rough modification has the side effect of doubling the movements of all the objects on the screen, not just the spaceship, it can be regarded as a successful attack. A better attack may consist of cloning all the functions involved in the movement, in order to call the tampered movement functions only for objects of type player, whose pilot is function `keyboardPilot`[2].

Hence, from the defender perspective, the security property to ensure on the `movePoint` asset is the *integrity* of behaviour of such function, as we want to avoid that the attacker can alter the way this function moves the spaceship controlled by the user.

In addition to the attack reported in Fig. 2, another easy way to compromise the unprotected asset `movePoint` consists of calling twice any of the movement functions (i.e., `movePoint`, `moveSolidPlayer`, and `movePlayer`) at every space ship movement. However, every call to `movePoint` (direct or indirect, through the `moveSolidPlayer`, or `movePlayer`) must be doubled to avoid inconsistencies and application failures, and to be considered a successful attack.

## 2.2 Client/Server Code Splitting

Client/Server Code Splitting is intended as a means to protect software by identifying portions of code that implements sensitive computation and that manipulates sensitive values, slicing these portions away, and moving them from an untrustworthy client to a trusted server, in order to prevent any tampering attack.

Moving entire functions is not always feasible, either because a function might be larger than the portion of code that needs to actually be transferred, or because of the side effects that might arise on the program execution correctness. For instance, the function to move might need to modify some global variables that should remain on the client. Therefore, to ensure that the original functionalities are unmodified in the new, protected configuration, data and control dependencies should be carefully taken into account.

The *barrier slicing* algorithm can be applied to limit the portion of code to be moved to the trusted server [6]. The developer is supposed to drive the protection process, depending on the security and protection requirements of the program. In our case, code annotations shall be used to annotate (i) the extent of the security-sensitive portion of the

[2] Interested readers can find details about these functions in the file `gamespace.c`.

code that needs protection; and (ii) the variables meant to store security-sensitive data, such that the code that handles their value is protected.

Then, the protection tool is applied. In our Client/Server Code Splitting prototype, annotations are read from the source code and used to compute the static slice by means of CodeSurfer[3], a commercial static analysis tool. Based on CodeSurfer output, an automated program transformation is applied to change the client code and remove the slice to be run at server side.

Furthermore, moving a portion of the client to the server introduces the need for additional communication, in order to exchange values between client and server, and to synchronize the execution of the sliced code. Communication works in both directions, since the server may require values from the client to keep the execution of its slice synchronized, while the client needs the values computed by the slice moved to the server and removed from the client code.

### *2.2.1 Barrier Slice Computation*

A barrier slice is based on the concept of (backward) slice [26]. Let $C = (x, V)$ be a slicing *criterion*, where $x$ is a statement in a program $P$, usually represented by the line number, and $V$ is a subset of variables in $P$. A backward slice $s$ on a given criterion $C = (x, V)$ can be defined as the sub-program $P'$ which behaves equivalently to the original program $P$ with respect to the criterion (i.e., for the computation of the values of variables $V$ at statement $x$). The slice $s$ on criterion $C$ can be computed as all the statements that directly or indirectly hold data or control dependencies on the variables in $V$ at statement $x$.

A barrier slice [18] is a backward slice on program $P$, where some statements are considered as *barriers*: the computation performed at those statements is considered not relevant for the slice from a security viewpoint, so such barriers are excluded from the slice and they block the propagation of data or control dependencies when the slice is calculated. A barrier slice can be computed by stopping the computation of a regular backward slice whenever one of the barrier statements is reached.

As proposed by Krinke et al. [18], let the System Dependence Graph (SDG) of program $P$ be an abstraction of the program where nodes represent statements and predicates, while edges carry information about control and data dependencies between statements. Then, given the program SDG $G = (N, E)$, the barrier slice $Slice\#(C, B)$, for the slicing criterion $C = (x, V)$ with set of barrier nodes $B \subseteq N$, consists of all nodes on which $x$ transitively depends over any inter-procedurally realizable path that does not travers any node of $B$, as follows:

$$Slice\#(C = (x, V), B) = \{m \in N | \begin{array}{r} p \in m \rightarrow_R^* x \wedge \\ p = \langle n_1 ... n_l \rangle \wedge \\ \forall 1 \leq i \leq l : n_i \notin B \end{array} \} \quad (1)$$

This definition can be easily generalized to a slicing criterion that involves a set of statements $X$, instead of a single statement $x$, assuming that the relevant variables $V$ are all sensitive variables used by statements in $X$. Hence, in the practical usage of barrier slicing we often refer to a set of statements $X \subseteq N$ and, with some notation overload, the slicing criterion becomes $C = (X, V)$.

### *2.2.2 Code Transformation*

As anticipated in section 2.2, to apply Client/Server Code Splitting, the developer first has to annotate her source code. Code annotations specify the slicing criterion $C = (X, V)$ as a set of statements $X$ and a list of sensitive variables $V$ on which to apply the barrier slicing algorithm, as well as a set of barrier statements $B$ that block the propagation of the dependencies while calculating the slice. Fig. 3 shows two annotated versions of the `movePoint` function that have been used as our experimental treatments (see Section 3.6). In Fig. 3a code is annotated to move the whole switch statement (medium-size split) to the server, while in Fig. 3b code is annotated to move just the code of each case statement. In Fig. 3a, the fields of variable `point`, i.e., `point->row` and `point->col` are specified as the sensitive variables to protect (line 10). The annotation to achieve this is: `_Pragma("begin_criterion(var=point->row, point->col)")`. With this annotation, any computation that affects them will be removed from the client code, until the barrier is encountered in the backward traversal of dependencies. In turn, the barrier is defined as any statement that assigns a value to variable `direction` (see annotation at line 6: `_Pragma("begin_barrier(var=direction)")`).

Fig. 4 and 5 report the results of the transformation, i.e., the sliced code with the necessary communication constructs. The client code, to be executed on the user platform, is reported on the left-hand side, and the server code, to be executed in the proper server thread, is on the right-hand side. The transformation consists of the following steps:

- *Remove defs.* Definition of (i.e., assignment to) sensitive variables that occur within the slice are removed from the client code, they are replaced with synchronization points because the computation at client and server side can proceed only after they both reach the same point. Hence, the removed statement is replaced by a `sync` statement (line 12, 17, 18, 24, and 25 in Fig. 4a). A `sync` statement is added also at the server side after each definition of a sensitive variable.

[3] GrammaTech CodeSurfer https://www.grammatech.com/products/codesurfer

```
1  void movePoint(Point* point, int
       direction){
2     assert (point != NULL);
3     if(direction==STOP){
4        return;
5     }
6     _Pragma("begin_barrier(var=direction)")
7     switch(direction) {
8     _Pragma("end_barrier")
9
10    _Pragma("begin_criterion(var=point->row,
          point->col)")
11    case N:
12       point->row--;
13    break;
14 // omitted as it is the original code
15    case NW:
16       point->row--;
17       point->col--;
18    break;
19    _Pragma("end_criterion")
20 }
```

(a) Annotated code: medium-size split.

```
1  void movePoint(Point* point, int direction){
2     assert (point != NULL);
3     if(direction==STOP){
4        return;
5     }
6     _Pragma("begin_barrier(var=direction)")
7     switch(direction) {
8     _Pragma("end_barrier")
9
10    case N:
11    _Pragma("begin_criterion(var=point->row)")
12       point->row--;
13    _Pragma("end_criterion")
14    break;
15    case NE:
16    _Pragma("begin_criterion(var=point->row,
          point->col)")
17       point->row--;
18       point->col++;
19    _Pragma("end_criterion")
20    break;
21   // similar cases omitted
22    case NW:
23    _Pragma("begin_criterion(var=point->row,
          point->col)")
24       point->row--;
25       point->col--;
26    break;
27    _Pragma("end_criterion")
28 }
```

(b) Annotated code: small-size split

Fig. 3: Source code of SpaceGame with annotations for the two treatments used in this experiment.

```
1  void movePoint(Point* point, int
       direction){
2
3    assert (point != NULL);
4
5    if(direction==STOP){
6      return;
7    }
8    send_initial_message();
9    send_value_to_server(1, point->row);
10   send_value_to_server(2, point->col);
11   send_value_to_server(3, direction);
12
13   switch(direction) {
14     case N:
15       synch_with_server(1);
16     break;
17     // cases omitted
18     case NW:
19       synch_with_server(11);
20       synch_with_server(12);
21     break;
22   }
23   point->row = ask_value_from_server(1, 1);
24   point->col = ask_value_from_server(2, 2);
25 }
```

(a) Client-side code.

```
1  void * execute_slice(){
2    check_send(1);
3    check_send(2);
4    check_send(3);
5    switch(direction) {
6      case N:
7        check_synch(1);
8        row--;
9      break;
10     // cases omitted
11     case NW:
12       check_synch(11);
13       row--;
14       check_synch(12);
15       col--;
16     break;
17   }
18   check_answer(1, 1);
19   check_answer(2, 2);
20 }
```

(b) Server-side code.

Fig. 4: Result of Client/Server Code Splitting transformation of the `movePoint` function (small-size split).

- *Remove uses.* Uses of sensitive variables occurring within the slice are removed. At the client side, they are replaced by an `ask` statement, to query the value of the variable that is computed at the server side (lines 23, 24 in Fig. 4a, lines 11,12 in Fig. 5a). At the server side, a `send` is added before each of these statements (lines 7, 12, 14 in Fig. 4b), to deliver the variable value to the client.
- *Barrier variables.* Even if they are not moved to the server, the actual values of barrier variables are needed at the server side, to correctly initiate the execution of the slice. At the client side, upon definition of barrier variables, a `send` statement is added (lines 8-11 in Fig. 4a, lines 6-9 in Fig. 5a), to communicate the variable value to the server. At the server side, a `receive` statement is added (line 2-4 in in Fig. 4b and in 5b).

At the server side, a component implements a main thread that manages connections and messages from and to the connected clients. The sliced code is executed in an auxiliary thread that is launched from the main one whenever a client

```
1  void movePoint(Point* point, int
       direction){
2    assert (point != NULL);
3    if(direction==STOP){
4      return;
5    }
6    send_initial_message();
7    send_value_to_server(1, point->row);
8    send_value_to_server(2, point->col);
9    send_value_to_server(3, direction);
10
11   point->row = ask_value_from_server(1, 1);
12   point->col = ask_value_from_server(2, 2);
13
14 }
```

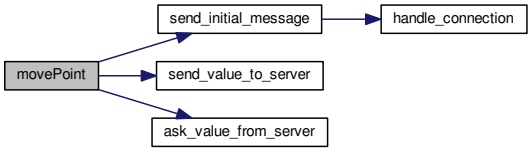
(a) Client code.

```
1  void * execute_slice(){
2     check_send(1);
3     check_send(2);
4     check_send(3);
5  //criterion for conf 1, point->row,
       point->col
6     switch(direction) {
7     case N:
8        row--; break;
9     // other cases omitted, as in the
         original program
10    case NW:
11       row--;
12       col--; break;
13 }
14 check_answer(1, 1);
15 check_answer(2, 2);
16 }
```

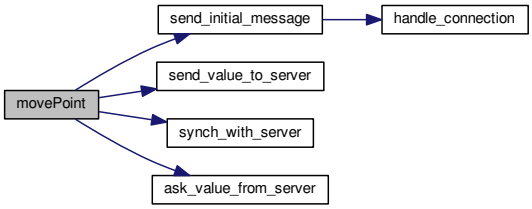
(b) Server code.

Fig. 5: Result of Client/Server Code Splitting transformation of the `movePoint` function (medium-size split).

connects. Function calls to instantiate the communication framework are also automatically added to the sliced code at the server side.




(a) Zoom into the `movePoint` call graph after the application of Client/Server Code Splitting (medium-size split).




(b) Zoom into the `movePoint` call graph after the application of Client/Server Code Splitting (small-size split).

Fig. 6: Call graph of the `movePoint` function after the application of Client/Server Code Splitting.

As a consequence of these transformations, Client/Server Code Splitting makes the call graph more complex (see Fig. 6) and removes some statements from `movePoint`. In particular, the medium-size split (see Fig. 6a) moves slightly bigger code blocks and requires less synchronizations than the small-size split, which moves smaller code blocks to the server and requires more synchronizations (see Fig. 6b).

Mounting an attack on the protected applications requires performing code changes that are equivalent to the ones presented in Fig. 2, despite the removal of the statements that compute the values of sensitive variables from the client. For instance, doubling the value of variables `point->row` and `point->col` requires that the attacker has understood or guessed when the server is actually computing and communicating such values to the client. Since such sensitive variables are managed and updated on the server, the attacker should change the calls to the server (e.g., by incrementing the value of the coordinates by 1 before sending them to the server) or should modify the values received from the server (e.g., by incrementing the value received from the server by 1). The attack that doubles the calls to function `movePoint` is also possible, but guaranteeing that the application does not crash becomes much more complex and time consuming, because it requires non trivial guesses on the server behaviour (as confirmed by the results of our experiment).

Note that, even if the code moved to the server can not be tampered with by an attacker, attacks to the application are still possible. This happens because, when the server-side computation is over and some results or data are sent to the client, the client code that uses these data can be still tampered with, to change how data are used. However, this attack is expected to be quite complex, because it requires the attacker to guess (at least partially) the hidden server-side computation.

## 3 Experimental Design

### 3.1 Goals and Research Questions

The main *goal* of the study is to evaluate the degree of protection offered by the Client/Server Code Splitting technique. The *purpose* is evaluating the ability of Client/Server Code Splitting to make the code resilient to malicious tampering

attacks. The *quality focus* is the ability of the technique to reduce the attacker's capability to successfully mount an attack, by making the understanding and modification phases more complex, due to missing code portions. The study evaluates the *perspective* of an attacker, as we aim at evaluating if there is an increase in the difficulty experienced by the attacker when the code is protected by Client/Server Code Splitting. In our case the role of the attacker is played by a group of students who have a consolidated minimum level of expertise in manipulating application source code.

The above goal can be achieved by means of an experiment aimed at answering the following four research questions:

- **RQ1 (correctness)**: How effective is code splitting for preventing code tampering as compared to the unprotected code? Does the level of splitting (medium vs. small-size splitting) affect the attack correctness?
- **RQ2 (time)**: How effective is code splitting for increasing the attack time as compared to the attack time for the unprotected code? Does the level of splitting (medium vs. small-size splitting) affect the attack time?
- **RQ3 (strategy)**: Does the attack strategy affect the attack effectiveness (considering both correctness and time)? Does the level of splitting (medium vs. small-size splitting) affect the chosen attack strategy?
- **RQ4 (process)**: What is the attack process followed by the subjects? Does it differ between successful and unsuccessful attacks? Is it influenced by the level of protection?

Answering the above questions will help software developers make informed decisions on how to protect their applications and what expectations can be reasonably met when using Client/Server Code Splitting.

The first research question is about the effectiveness of Client/Server Code Splitting as a code protection technique intended to reduce the likelihood that attackers complete a successful attack. We investigate this research question by differentiating the correctness by level of splitting (medium vs. small-size splitting).

In several real world scenarios, defenders can be satisfied if hackers are not able to mount their attack in a given limited time frame (e.g., between two consecutive releases of the software). Correspondingly, the second question focuses on the ability of Client/Server Code Splitting to delay attackers from mounting successful attacks. Indeed, like other protections, Client/Server Code Splitting comes without any assurance of being able to completely avoid attacks. It represents a hurdle meant to slow down attackers. In fact, delaying attackers is the key to render the attack not economically convenient. Again, we investigate this research question by differentiating the time effectiveness of Client/Server Code Splitting by level of splitting (medium vs. small-size splitting).

The third research question is meant to investigate the relation between attack strategy and attack effectiveness, where the attack strategy is simplified into three alternative high-level approaches to code tampering: (1) the sensitive asset is attacked directly; (2) the sensitive asset is attacked indirectly (e.g., by manipulating functions that call the function containing the asset); (3) a mix of (1) and (2). We also investigate whether there is any relation between the level of protection and the attack strategy adopted by the experimental subjects, to understand if splitting smaller/larger code portions triggers different strategies. Such knowledge is fundamental to tune Client/Server Code Splitting techniques and to design new techniques.

The fourth research question is about the process followed by attackers. We answer this research question qualitatively, by applying a process mining tool (namely, Disco[4]) to the annotated attack reports collected from the attackers, and by inspecting the recovered attack processes. In particular, we want to understand if attackers who complete their task successfully follow a process that differs from the one followed by unsuccessful attackers. We also want to understand if the level of protection influences the followed attack process.

### 3.2 Threat Model

As anticipated, in this empirical investigation, we consider the scenario known as Man at the End (MatE) [13]. This scenario assumes that the program to protect will be running on an untrustworthy device that is under full control of the attacker. The program developer cannot, by any mean, control the execution environment where programs will be executed. There are no assumptions on the execution environment; it could be a general purpose computer or a dedicated device, e.g., a smartphone. The attacker can perform sophisticated operations like (among others):

- access to the software binary code;
- own and use a broad range of tools (e.g. debuggers, decompilers, disassemblers, fuzzers) to perform (static, dynamic, hybrid) analysis of the program;
- monitor its execution access to the program memory and extract any kind of valuable information;
- tamper with the program by means of static changes of the executable program (crack, patch);
- tamper with the program by dynamically injecting code while it runs (e.g., by means of a debugger).

Moreover, the attacker can perform all the types of Man-in-the-Middle attacks, both passive and active ones (e.g., sniffing, spoofing, forging, reply attacks). Nonetheless, with proper reverse engineering effort and dynamic analysis, the attacker can also decrypt communications protected with channels considered secure against MitM attacks, as he can tap processes that process data after decryption.

[4] https://fluxicon.com/disco/

On the other hand, the MatE scenario assumes that servers under the control of the defenders (program developers of client-server programs, servers devoted to software protections) are trustworthy. That is, servers cannot be accessed, analyzed by the attacker, because he has not valid credential to access it. Moreover, active attacks against the server and exploitation of the server vulnerabilities (e.g., cross-site scripting, SQL-injection attacks, CVE exploits, vulnerability assessment) are considered outside the scope of the MatE and of the threat model we intend to investigate in this paper.

Moreover, the attacker has no limitations on the attack strategy, like timing and planning of the attack. He can gain, by any mean and analysis, valuable information and elaborate and mount tampering task in any order.

In short, in our experiment, we focus on a client-server architecture where the attacker has full access to the client part only. The server executes, together with the slices of code extracted from the program, in a host that cannot be attacked.

Finally, we are interested in studying how hard it is to understand and tamper with a program. We are not interested in evaluating how hard it is to decompile or disassemble compiled code, because there are quite good tools to (at least partially) automate this task[5]. Thus, even if the source code might not be immediately available to an attacker in an intelligible form (e.g., because the program is a compiled binary machine code or it has been obfuscated), we assume that source code almost as comprehensible as the original source could be obtained with proper effort. Thus, we assume that the actual attack starts after decompilation tools already succeeded and some form of source code is available to attackers.

### 3.3 Subjects

The *context* of the study consists of *subjects*, i.e., the students acting as attackers, who perform their attacks on *object*, i.e., the system to be attacked.

Subjects are 87 Master students in Computer Science Engineering from Politecnico di Torino selected from the students attending the "Computer and System Security" course, which is held in the second and last year of the Master. Politecnico di Torino offers and requires a strong knowledge of C programming in several courses. Indeed, all the Computer Science Engineering students have to attend several courses on C programming (as well as OO programming and scripting) during their Bachelor and during the first year of their Master[6]. Moreover, for students enrolling in the Master from a different university, Politecnico di Torino ensures that they have equivalent competences in C programming (by checking their careers and exams). Otherwise students are forced to add undergraduate courses to their curricula, to compensate their lacks. Therefore, we have estimated the background in C programming of all the students attending those courses enough to perform the experiment. That is, all the students were eligible as subjects. Nonetheless, to select the subjects based on their motivation, the participation to the experiment was on a voluntary basis. All the participants that performed the experimental task with diligence (regardless of the attack success) were rewarded with 2/30 extra points in the final course exam. In general, we can assume that subjects have a background in computer network security because, during the period of the experiment, they where following the Computer and Security analysis course. This course covers both theoretical and practical levels of all the basics of ICT security and risk analysis, cryptography, authentication systems, X.509, PKIs and e-documents, security of IP networks and network applications (web, file transfer, etc.), firewall and IDS/IPS, e-mail security[7].

Furthermore, we have evaluated their exact C skills before and during the experiment. In particular, before the experiment we asked the participants to complete two online C tests. Both tests were multiple-choice tests, the first[8] consisting of 10 questions, and the second[9] of 20 questions. These tests include questions intended to check knowledge of theoretical aspects of C programming (e.g., use of pointers, static variables, standard C libraries) and the behaviour of compilers (e.g., call to functions, variables initialization, stack overflows). However, most of the questions were aimed to check code comprehension abilities (e.g., by asking the expected output of a small piece of code). Even if a proposing a programming exercise can be considered an alternative approach to estimate the programming abilities of students, we have preferred an approach based on multiple-choice test. Indeed, we were not interested in assessing the abilities of the subjects in writing code that implements a set of, maybe complex, functionalities, which is important when hiring programmers. Given the attack task we were asking to accomplish (a comprehension task to decide where to perform the changes and the attack strategy to achieve the attack goal followed by a limited number of code modifications), we were more interested in evaluating the abilities of the students in comprehending and mentally simulating pieces of code.

The global results of the two tests are reported respectively in Fig. 7a and in Fig. 7b. To confirm such results (and try to identify students that could have cheated during the first set of tests answered at their place), we asked the subjects,

[5] Two examples of excellent reverse engineering and exploitation tools are IDA-Pro `https://www.hex-rays.com/products/ida/` and radare2 `https://rada.re/`

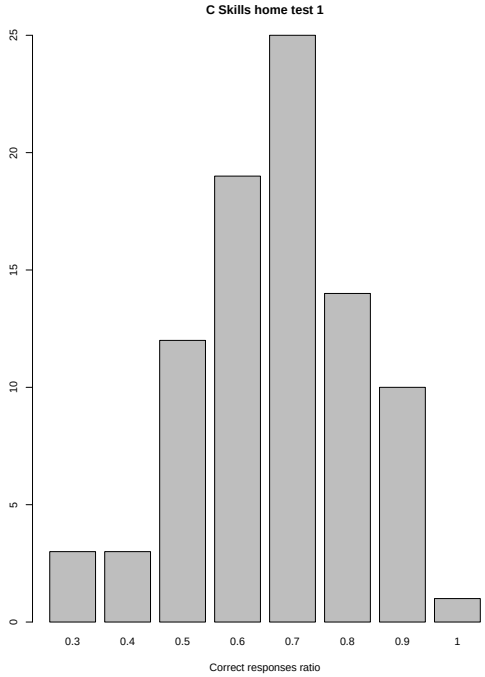
[6] The syllabus reports the following course that "Computer and System Security" students have followed: Computer Science (introduction to C programming), Algorithms and Programming, Object oriented programming, Operating systems (scripting languages), at Bachelor level, and System programming, Distributed Programming I and for the Master together with the optional course named Distributed Programming II.

[7] `http://security.polito.it/~lioy/02krq/`

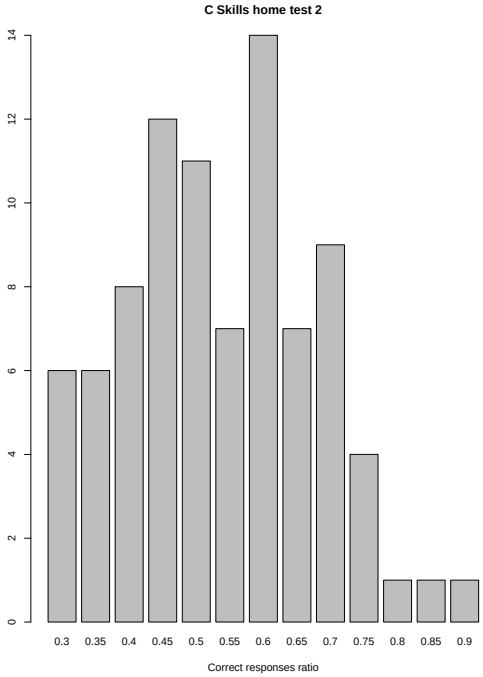
[8] `https://www.proprofs.com/quiz-school/story.php?title=test-your-c-skills`

[9] `http://www.pskills.org/c.jsp`

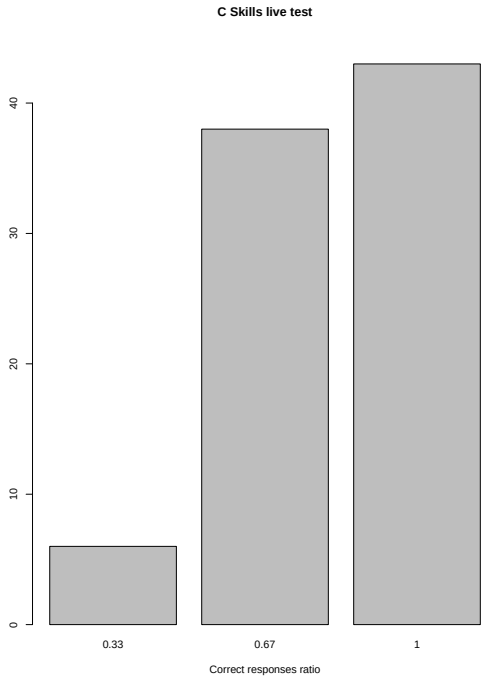
at the start of the experiment, to answer three more multiple-choice questions, whose results are summarized in Fig. 7c. Furthermore, Fig. 7d reports the composite results of the two home tests and the additional questions made at the start of the experiment.



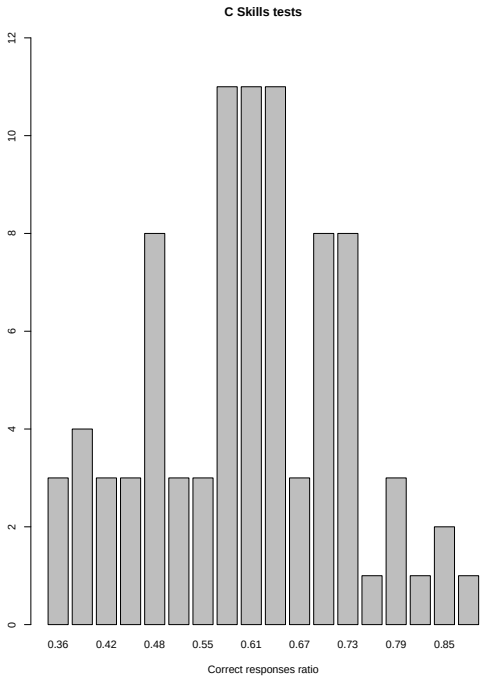

(a) Global results of the first home test (10 questions)




(b) Global results of the second home test (20 questions)




(c) Global results of the additional questions (3 questions)




(d) Composite results of all the tests (33 questions)

Fig. 7: Global results of the preliminary tests.

Subjects are not expected to have any knowledge about MatE scenarios, attacks, and attack strategies. Indeed, the involved students did not attend any course about software tampering or software reverse engineering.

From the statistical information we have from the student office of the Politecnico di Torino, we know that all the participants have at least 1 year of experience, however, the great majority of them has at least 2 years of programming experience in C. Indeed, some students in the Computer Science Engineering have experience either as part-time or

as full-time professional programmers. This information is also confirmed by the answers collected with an optional post-experiment question (see the Appendix).

Nevertheless, for the purposes of our experiment, we ensured, by asking with a preliminary questionnaire, that all the participants were able to use a C debugger to set break-points, do stepwise execution, inspect the call stack and inspect program variables. All the students reported that they had good experience with at least one of the supported IDEs (Visual Studio, CodeBlocks, XCode, Eclipse). Moreover, two students declared that they would have preferred to compile and debug with a command line approach (gcc + gdb). Note that both command lines tools and all the IDEs listed are theoretically perfectly suited for performing the attack task we proposed as well as the operating systems to use. Therefore, subjects were allowed to choose their preferred attack environment. Nonetheless, tools can have an influence on the result of the tasks, as discussed in Section 3.11.

### 3.4 Object

The object of this experiment is the open source program SpaceGame, already presented in Section 2. The framework and the demonstrator amount to 1.873 SLoC (measured by `sloccount`), including header files. SpaceGame can be run in interactive and in batch mode, by passing in a command line string with the input keys to execute. For instance, running the SpaceGame executable with the following `./game jjll` starts the game then moves the spaceship twice to the left then twice to the right. When executed in batch mode, SpaceGame prints the sequence of coordinate changes associated with the input keys.

Even if GAME was originally created for Unix platforms, it can be ported to more systems because it uses standard ANSI C. Therefore, we were able to support different platforms, so as to allow subjects to work on their preferred environment.

### 3.5 Attack task

As anticipated in Section 2.1, the attack task to be executed by the experimental subjects on SpaceGame aims at simulating an attacker gaining an unfair advantage over other competing players:

> *Modify the source code of SpaceGame so that every key press moves the spaceship twice as fast as allowed by the game rules. Specifically, in the GAME context, each key press must translate into a 2-character length move, instead of a 1-character move.*

A successful attack task was determined by the execution of a standard assessment procedure, defined as a set of sequences of key presses passed via batch mode to the program. To avoid unfair behaviours (e.g., subjects trying to only solve the problem checked by the assessment procedure, instead of mounting the complete attack task) the procedure was only known to the assistants.

### 3.6 Treatments

To evaluate the impact of different ways of splitting the code on the level of protection, we considered two treatments corresponding to the two splitting configurations described in Section 2.2.2. The two configurations have the same barriers but different criteria (see Fig. 3). The first configuration, named $T_S$ (small-size splitting treatment), is characterized by multiple and small portions of code as criteria, in particular four criteria containing 1 statement and four criteria containing 2 statements. For the second treatment, named $T_M$ (medium-size splitting treatment), a single but larger portion of code (28 statements) is annotated as unique criterion. Of course, there is also the baseline treatment $T_C$ (clear code treatment), which is the original SpaceGame code.

Different annotation configurations require the Client/Server Code Splitting transformation to move smaller/larger portions of code from the client to the trusted server and to exchange more/less messages between client and server. Table 1 reports the size of the criteria and the size of the source code for the three different versions of SpaceGame used in the experiment. As anticipated, the original game (first line) consists of 1,873 lines of C code, measure with *sloccount*. The treatment with small-size criteria (1 or 2 lines) results in a larger client code (1,880 lines) than the treatment with a medium-size criterion (1,850 lines). The corresponding slice on the server side is also larger (45 lines for $T_S$ and 34 lines for $T_M$). Consequently, the overall server, including the framework and the slice, is also bigger (469 lines for $T_S$ and 457 lines for $T_M$).

The difference in size is mostly due to the difference in the number of data dependencies that are cut by the slice $T_S$ vs. $T_M$. The more the data dependencies between the slice to move on the server and code left on client, the more the communication between the server and the client needed to propagate updated values. As shown in the last column of Table 1, while in the code of $T_M$ contains just 5 split communication messages (3 *send* and 2*ask*), the code of $T_C$ contains 17 messages (3 *send*, 12 *sync* and 2*ask*).

There are conflicting effects in choosing the configuration of the slices to move to the server. On the one hand, the larger the portions of code executed on the server the larger the server execution overhead. This effect is dangerous especially when serving many clients at the same time. On the other hand, splitting several small portions of code may require frequent synchronizations between the code remaining on the client and the code moved to the trusted server, which may introduce a possibly unacceptable communication (hence, performance) overhead. For this reason, it is interesting to investigate both a small-size and a medium-size code splitting treatment.

Table 1: Size of clear and obfuscated code, measured with sloccount

| Version | Criterion size | Component | | | Messages | | | |
|---|---|---|---|---|---|---|---|---|
| | | Client | Slice | Server | Send | Sync | Ask | TOT |
| $T_C$ Original | - | 1,873 | - | - | - | - | - | - |
| $T_S$ Split small | 1 x4, 2 x4 | 1,880 | 45 | 469 | 3 | 12 | 2 | 17 |
| $T_M$ Split medium | 28 | 1,850 | 34 | 457 | 3 | 0 | 2 | 5 |

We report in Table 2 a set of metrics of the SpaceGame program, computed with Radare2[10] that allow a better estimation of the size and complexity of the object of our experiment.

### 3.7 Experimental procedure

In this section we provide a detailed account of the procedure followed during the experiment and of the materials given to the subjects. The experimental procedure is composed of three main phases:

1. preliminary information gathering;
2. controlled experiment;
3. post-experiment information gathering.

**Preliminary information gathering.** We decided to let the subjects use their PC to carry out the experiment, in order to have them execute the task in the most familiar setting, which is also closer to happens to real hacker. However, to minimize the experiment setup time for the subjects, and to avoid time wasted due to simple compilation or execution errors (e.g., caused by misconfiguration of IDE projects), we performed a preliminary information gathering phase, where we collected data about operating system and the IDE subjects were more familiar with, as well as their second choice. We used this information to identify the most commonly used combinations of OSs and IDEs, and we created, for each combination, three different IDE projects, one per treatment. Specifically, we supported the following OS and IDE combinations:

- Windows: CodeBlocks, Microsoft Visual Studio 2013, Microsoft Visual Studio 2015;
- Mac OS: XCode;
- Linux: CodeBlocks, gcc (we provided a Makefile and tested source code debugging with gdb).

We supported these three IDEs because they covered more than 93% of the participants (81 out of 87 participants). A limited number of students who did not indicate any of the three supported tools either as first or as second choice, received support prior to and during the experiment, to install a different IDE or to adapt the experimental material to their environments. None of them reported problems in their post-experiment reports. The IDE projects for Windows use a porting of NCurses to Windows, part of the Public Domain Curses library. Since NCurses is used in SpaceGame only for visualization purposes, and since its code is not involved in any part of the attack task, this difference from the other IDE projects does not hamper in any way the experiments results.

**Controlled experiment.** At the beginning of the controlled experiment, the following materials were distributed to the participants:

[10] https://github.com/radare/radare2

Table 2: Metrics of the programs used for the three treatments.

| Treatment | # functions | # basic blocks | # edges in the CFG | Ciclomatic complexity | # calls | # ASM instructions | # conditional jumps | # jumps |
|---|---|---|---|---|---|---|---|---|
| $T_C$ | 92 | 717 | 858 | 635 | 221 | 5179 | 259 | 402 |
| $T_S$ | 101 | 772 | 913 | 677 | 282 | 5773 | 280 | 431 |
| $T_M$ | 101 | 769 | 915 | 667 | 278 | 5763 | 279 | 427 |

- a brief skill test on C programming, composed of three questions, to confirm the results of the preliminary questionnaires undertaken at home by the subjects during the preliminary information gathering;
- a description of SpaceGame, with instructions on how to execute and interact with it;
- a link to a remote folder containing all IDE projects for the assigned treatment;
- a detailed description of the procedure to follow to execute the attack task.

During the controlled experiment, we asked the subjects to execute the attack task by following the procedure stated below:

1. undertake the brief C skill test;
2. read the provided description of SpaceGame;
3. download the archive containing the project for the preferred IDE and OS from the provided link;
4. extract the archive and open the project with the corresponding IDE;
5. compile the project;
6. execute the SpaceGame application obtained in the previous step; for treatments $T_s$ and $T_m$, before executing SpaceGame, launch the server application;
7. practice with SpaceGame, without looking at the source code, in order to understand how it works at high-level;
8. read the description of the attack task to execute (provided in Section 3.6);
9. write down the start time for the execution of the task;
10. execute the task;
11. temporarily annotate the end time for the execution of the task;
12. prove to one of the assistants that the attack task was completed correctly;
13. if the task is approved by the assistant, write down the definitive end time or continue the attack task from step 10.

**Post-experiment information gathering.** After the experiment, i.e., during the post-experiment information gathering, we asked students to:

1. write and send by email a detailed report on how the task was executed, including the attack steps, the tried attack strategies, and, if known, the time devoted to each activity;
2. fill in an optional post-experiment questionnaire, by which we gathered additional information on the subjects' past programming experience and feedback on the experiment.

Throughout all the experimental phases, we provided assistance to the subjects and answered all their questions but those related to ways to successfully carry out the attack task. Indeed, when setting up their environment, and despite the testing we performed on all platforms, we had also to solve some setup issues.

The gathered reports, with the results of the C skill questionnaires and the time needed for the execution of the attack task, constitute the data on which we base our empirical analysis.

### 3.8 Variables

The selection of variables collected during the experiment and the formulation of hypotheses have been driven by the research questions defined in Section 3.1.

As *dependent variables*, we consider the following aspects of the executed attack tasks:

**Correctness**: evaluates whether the participant correctly performed the task, i.e., whether the attack was successful or not, as assessed by one of the assistants. The variable is defined as:

$$Corr(s_i) = \begin{cases} 1 & \text{if subject } s_i \text{ succeeded in performing the attack task} \\ 0 & \text{if subject } s_i \text{ failed.} \end{cases}$$

**Time**: represents the elapsed time used to perform the attack task. Variable $Time(s_i)$ measures the minutes spent by subject $s_i$ to perform the attack task, regardless of the success of the attack itself.

**Strategy**: indicates the category of attack strategy tried by the participant to mount the attack.

In order to identify the attack strategy we analysed the reports gathered from the participants after the experiment. In particular, we adopted a bottom-up approach consisting of the following procedure:

- the authors independently analysed the reports and tagged them with labels describing the attack;
- in a joint meeting the authors then merged similar tags; and,
- they identified categories by grouping together related attack approaches.

The above procedure led us to define three main categories of attack strategies:

*internal*: the participant altered the computation inside functions containing sensitive assets, e.g., by changing the content of the `movePoint` function;

*external*: the participant changed the actual parameters of a function call or altered the call chain eventually leading to invoke a sensitive function, e.g., by calling twice all the functions that manipulate the space ship movement;

*mixed*: the participant applied a strategy that is the combination of the two previous strategies.

Note that time and correctness are related to the duration of the experiment, i.e., the quantity they measure is not independent from the time (2h) we have given the participants for the task. That is, one would expect the more the time available to complete the attack task, the more the participants who can correctly complete it, and the higher the average time. Such relationship has been considered during the design and does not affect the results of our study (treatments have been fine tuned to allow at least some participants to succeed in the two hours available for the controlled experiment), since the effectiveness of Client/Server Code Splitting is assessed by observing the dependent variables on clear vs. obfuscated programs in equivalent conditions.

As *independent variables*, we consider the following:

**Treatment**: indicates the type of obfuscation applied to the source code; the type of Client/Server Code Splitting applied is a value from the set $\{T_C, T_S, T_M\}$, as described in Section 3.6.

**C.SCORE**: indicates the C languages skills of the subjects. The score is based on the aggregate results of the offline (pre-experiment) and online (during the experiment) tests. It ranges between 0 and 1.

## 3.9 Hypotheses

Based on the research questions RQ1, RQ2, RQ3 and the variables defined above we can formulate the following null hypotheses to be tested:

- $H_1$: at each level of protection, Client/Server Code Splitting has no effect on the correctness of an attack.
- $H_2$: at each level of protection, Client/Server Code Splitting has no effect on the time to perform an attack.
- $H_{3a}$: the attacker's strategy has no effect on the effectiveness of the attack (both in terms of correctness and time).
- $H_{3b}$: the level of Client/Server Code Splitting has no effect on the strategies the attackers use to mount an attack.

Being a qualitative research question, RQ4 is not associated with any formal null hypothesis.

## 3.10 Analysis method

### *3.10.1 Hypothesis testing*

The experimental measures are first summarised with basic descriptive statistics. Correctness is reported in terms of both absolute numbers of correct answers and proportion. For Time we report the mean and standard deviation.

The former two hypotheses will be tested using a frequentist hypothesis testing approach considering first the main factor and then the C skill as co-factor.

The attack strategy will be analysed first as an independent variable, then as a co-factor influencing correctness and time; eventually we'll analyse how C skill influenced the selection of a strategy.

To test hypothesis $H_{01}$, concerning Correctness, we use a logistic regression of Correctness vs. Treatment. Such analysis is suitable for the dichotomous nature of the output measure. The logistic regression is based on the following model:

$$p_{Correctness} = \frac{1}{1 + e^{-(\beta_0 + \beta_{\text{medium}} \cdot T_{\text{medium}} + \beta_{\text{small}} \cdot T_{\text{small}})}}$$

where $T_{medium}$ and $T_{small}$ are indicator variables for the Split-medium or Split-small Treatment levels. In particular:

$$T_{medium} = \begin{cases} 1 & \text{if } Treatment = \textit{Split-medium} \\ 0 & \text{if } Treatment \neq \textit{Split-medium} \end{cases} \qquad T_{small} = \begin{cases} 1 & \text{if } Treatment = \textit{Split-small} \\ 0 & \text{if } Treatment \neq \textit{Split-small} \end{cases}$$

Looking at the exponent, we find the so called *logit* expression. We observe that, since the treatment can only assume one value at a time, at most one of the indicator variables at a time can be equal to one. Moreover, we observe that there is no explicit indicator variable (and relative coefficient) for the *Clear* level; such a level corresponds to both indicators being equal to 0, i.e. the intercept ($\beta_0$) of the *logit* expression.

The results of the logistic regression can be interpreted by recalling the definition of the *logit*() function for the probability of Correctness $p_{Correctness}$:

$$logit(p_{Correctness}) = log\left(\frac{p_{Correctness}}{1 - p_{Correctness}}\right) = log(\text{Odds})$$

Therefore each coefficient in the *logit* expression can be interpreted as the log-odds of correctness for the corresponding Treatment level; e.g. the intercept $\beta_0$ coefficient corresponds to the log-odds of Correctness when the *Clear* Treatment is applied.

In addition, keeping in mind the relationship between Odds Ratio ($OR$) of two condition (A and B) and the $logit$ values:

$$log(OR_{A/B})log(\frac{Odds_A}{Odds_B}) = logit(p_A) - logit(p_B) \quad \text{or} \quad OR_{A/B} = e^{(\text{logit}(p_A) - \text{logit}(p_B))}$$

we can compute the relative correctness odds ratios for any pair of Treatment levels.

To test hypotheses $H_{02}$ concerning Time, we conduct a non-parametric test equivalent to ANOVA – permutation test – of the output variable vs. Treatment, Correctness, and their interaction. The choice of a non-parametric test method is due to the expected non-normality of the measures. The linear regression is based on the following model:

$$Time = \beta_0 + \beta_T \cdot T + \beta_C \cdot C + \beta_{C:T} \cdot C{:}T$$

Where *C:T* represents the interaction between Correctness and Treatment.

Concerning the effects of the C skill co-factor we perform separate analysis – one for each Treatment level – for Correctness and Time. Logistic regression will be used for the former and a linear regression for the latter.

In order to test hypotheses $H_{03}$ concerning the attack strategy, we perform a $\chi^2$ test comparing the frequencies of strategies with different Treatment levels. We expect the external strategy to be more common for obfuscated code. Therefore, after identifying a significant effect, we apply a Fisher exact test to estimate the odds ratio of using an external strategy with the Clear treatment vs. either Split treatment.

In order to get more insights and understand how much a strategy actually paid off, we analyse the effect of strategy on both Correctness and Time. The former analysis is performed by means of a logistic regression, the latter using a non-parametric ANOVA with permutation tests.

The logistic regression is based on the following model:

$$\text{logit}(p_{Correctness}) = \beta_0 + \beta_{\text{External}} \cdot S_{\text{External}} + \beta_{\text{Mixed}} \cdot S_{\text{Mixed}}$$

where $S_{External}$ and $S_{Mixed}$ are indicator variables for the External or Mixed Strategy levels respectively.

Similarly to the previous logistic regression, since the strategy can assume only one value at a time, at most one of the indicator variables at a time can be equal to one; moreover, there is no explicit term for the *Internal* level, that is represented by both indicators being 0, i.e. the intecept ($\beta_0$) of the *logit* expression.

The linear model used for the permatation tests is

$$Time = \beta_0 + \beta_T \cdot T + \beta_S \cdot S + \beta_{S:T} \cdot S{:}T$$

where $T$ is the treatment and $S$ the strategy.

The assessment of the statistical test results is carried out assuming significance at a 95% confidence level ($\alpha$=0.05). When testing the effect of co-factor C skill, since we test three distinct hypotheses on the same set of participants, to avoid inflating the family-wise error rate we applied the Bonferroni correction; therefore we employ a corrected $\alpha_C = 0.05/3 = 0.017$ for decisions. So, we reject the null-hypotheses when $p-\text{value} < \alpha_C$.

All the data processing is performed with the R statistical package [21]. In particular the permutation test analysis was conducted using the lmPerm package [27].

#### *3.10.2 Attack Process Analysis*

Participants were asked to fill a report where they describe in free text the attack strategy followed during the execution of the attack task. To analyze the reports we annotate them using concepts taken from a closed taxonomy [9] and we apply process mining to automatically extract a process model from the sequences of annotations in each annotated report. In this way, we obtain a process model of the activities carried out by attackers. Moreover, we apply process mining after partitioning the reports, to analyze the occurrence of any difference in the followed process among the groups that form the partition. In particular, we apply process mining separately to the reports produced by participants working on clear and to those working on protected code, in order to understand if a different process is followed depending on the presence of a code protection.

The task of report annotation consists of the following steps: (1) read a sentence of the report to be annotated and if necessary split it into sub-sentences; (2) consult the concepts in the attack taxonomy to identify the concept that most accurately summarizes the activity described in the sentence; (3) if such a concept exists, add it to the report as annotation; (4) proceed to the next (sub-)sentence and repeat from step (1). To add annotations to sentences we used the comment function of the *Word* text editor. The taxonomy of concepts used to annotate the attack steps is the taxonomy that was obtained by studying three industrial cases, in which professional hackers performed attack tasks and reported their activities. The taxonomy is presented in detail in a previous ICPC paper [9]. The use of a closed taxonomy, as opposed to open annotations, reduces the effects of subjectivity during the annotation process.

The extraction of a process model from the set of annotated reports makes use of process mining. In particular, we use the DISCO[11] commercial tool for process mining. This tool supports a threshold on the activity frequency and a

[11] https://fluxicon.com/disco/

threshold on the path frequency. Such thresholds are set to avoid that infrequently executed activities or paths clutter the process diagram with irrelevant activity sequences, increasing the number of paths and making the diagram difficult to read and interpret. After a few preliminary trials, we have set the threshold for the activity frequency to 20% (i.e., the 20th percentile of most frequent activities is kept) and we have not set any threshold on the path frequency. The output of DISCO is an automatically inferred process model. The advantage of automating the process inference analysis step is that no subjectivity is involved in this step. The disadvantage is that the resulting process is obtained from a purely statistical analysis, which might overlook important semantic information that has low frequency but would be regarded as important by a human. Overall, we found the benefits of fully automated process mining to prevail over the disadvantages associated with subjective, manual process derivation from the annotated reports.

### 3.11 Threats to validity

We have checked our experiment against the checklist of the possible threats to validity proposed by Wohlin et al. [28], which are classified into construct, internal, conclusion, and external validity threats.

First, we present *construct validity* threats, which concern the relationship between the theoretical constructs and the actual metrics defined for the experiment.

- Client/Server Code Splitting is expected to affect both code comprehension and code change (i.e., the strategy). However, we could not measure in our experiment the actual level of comprehension subjects achieved when performing the attack tasks. Therefore, we assume subjects achieved the minimum comprehension needed to perform their attack tasks.
- We collected information about the time spent by the subjects to complete their attack tasks. Time is a coarse grained metric (minutes) that measures both comprehension and change activities. Though the two activities might in principle be separated, a typical attack consists of a close interleaving of the two. Moreover, it would be practically impossible for subjects to precisely assess the two times separately during the post-questionnaire (after a two hours experiment).
- Correctness of the attack task is evaluated as a boolean outcome. Although this is a very simple and crude metric, it reflects a real-case scenario where the attacker either gets access to the protected resources or not.
- All the possible strategies to mount the attack have been categorised into internal, external, mixed. This metric is quite coarse grained. However, it captures well the types of analysis and the modifications tried by the attackers during their task. We have analysed all the possible ways to perform the attack task against the SpaceGame application, all the identified ways could be easily mapped onto one of the three strategies.
- We measured the expertise in C programming with two C tests performed in an environment that was not controlled. The tests were complex and complete enough to ensure that the ability of students could be properly determined and using C programming tests is a standard practice for recruiting programmers used by several companies worldwide. We assured students that the C tests would not have been used for any activity not related to the empirical study and would not have been shown to the chair of the Computer System Security course. Moreover, we provided a third, shorter test to cross check the results of the preliminary C tests. A posteriori, after cross checking the results of the three tests, we have noted no major anomalies.
- We used an existing taxonomy to annotate the students' reports. We considered it complete enough to cover the tasks the subjects were asked to perform during the attack task. This taxonomy was built in a much more complicated context [9, 10], but once applied to the reports collected for this experiment, we found no need for concepts not included in the taxonomy. Therefore, closed coding using the existing taxonomy was deemed appropriate.
- To complete their attack task, participants were free to use any tool they know. This could be a confounding factor that may influence the results, because a better tool could help in completing the attack task faster and more precisely. We tried to control this threat by asking participants to report about their attack strategy, which could include tools.
- We used a process mining tool to extract process diagrams from annotated reports. While other process inference approaches could have led to different results, we found the process diagrams automatically produced by Disco to be meaningful and representative of the sequences of actions described in the reports. Hence, we think the qualitative analysis obtained from the mined processes was able to capture key properties of the followed attack procedures, although possibly not all properties that one might identify by other forms of qualitative analysis.

As second category of threats, we consider the *internal validity*, which is concerned with the capability to capture a cause-effect relation between the independent variables and the experiment outcomes. In this category, we consider all noise factors that may indirectly affect the outcomes and argue that they have been eliminated or measured (and assessed as negligible).

- From our past experience [9] we know that professional hackers use quite sophisticated tools to support understanding tasks, especially when protections are applied to the software. While we could not ask our subjects to use such tools, the need for them was limited since we provided subjects with the source code, usually unavailable to professional hackers.

- While attackers usually start from the compiled code, we gave source code to the participants. This represents a worst case scenario, where accurate reverse engineering tools are able to recover essentially the original source code. In reality, with proper effort an experienced hacker is able to reconstruct (almost) completely the source code. Decompiling is often a preliminary operation in several attack paths [9,10].
- While there are several types of attacks, we only focused on attacks that imply understanding and modification of the source code. While we cannot exclude that there could be different ways to mount attacks, alternative scenarios have not been tested.
- Before starting any activity, the tasks and attack objectives have been explained to all subjects. The post-questionnaires confirmed that no understanding issues manifested during the the experiments. Since the experiment has been conducted in a single session, we exclude all the threats related to time and repetitions (e.g., history, testing, mortality, statistical regression among experiments).
- The assignment of students to treatments has been randomly done. The reason is that subjects are all students of Politecnico di Torino, same year and same course thus we assumed their background as very homogeneous. Information about their programming skills has been used as co-factor. A posteriori, we verified that, even if their C skills were not homogeneous, the distribution of students to treatments was balanced.
- We encouraged the students' participation by granting 2/30 bonus[12] for the Computer System Security grade. We promised to all the participants that if they had completed with diligence all the tasks assigned to them, they would have been granted the bonus, regardless of the success in performing the attack task and regardless of the format and usefulness of the information provided in the report. Students were informed that proving that they did their best in performing the requested attack task was the criterion for receiving the bonus in advance. Nevertheless, we stimulated them to do their best in successfully completing the attack tasks.
  In our opinion, and from our experience with previous optional activities we organized with students, the bonus encouraged participation. Giving no incentives at all was not possible, as students do not usually like to spend extra time on non-profitable academic tasks instead of investing it in regular academic activities. Assigning the bonus to all participants led to a high participation, but introduced the risk of noise into the collected data, as some subjects might have only be interested in the bonus. Therefore, we added several checks both on the expertise, to identify subjects that did not score well on the C tests, and on the quality of reports, to identify subjects who did not document properly their activities.
- All students come from the same University and the same Master degree. Although at this point of their educational career they have been necessarily exposed to different programming languages, frameworks, styles and approaches, there might be a bias associated with the specific educational program adopted at Politecnico di Torino. Master students from a different University might possess a different background, resulting in different performance during the execution of the attack tasks.

*External validity* threats could make it impossible to generalise our results to the case of real attackers who want to compromising real applications protected by means of Client/Server Code Splitting.

- Professional hackers could be better subjects to evaluate MatE attacks exploitation, but it is considerably difficult to involve them. Nonetheless, the selection of participants was made on a voluntary basis: our assumption was that greater motivations could capture reasonably well hackers' profile. Certainly, students expertise in hacking programs is far from that of "professional" hackers. Nevertheless, the problem solving ability of the best students is not expected to be so different from that of hackers. Indeed, the level of expertise certainly affects the Correctness variable, especially because the attack task had to be mounted in a limited time frame. Therefore, we selected the objects and the treatments so that enough subjects could perform a successful attack in the time available for the experiments. Furthermore, our outcomes and conclusions on the effectiveness of Client/Server Code Splitting have been measured and estimated by comparing the performance of subjects on clear and obfuscated versions of the application. Hence, students with homogeneous expertise are expected to lead to the same validity of comparative results as hackers with homogeneous expertise. In other words, reporting a comparison with clear code mitigates to some extent the lack of hacking expertise. The debate on the use of students as proxies for professionals in empirical software engineering has been active for a long time and has recently attracted increasing attention [14]. For laboratory experiments, needed to address research questions that require a very controlled setting, it can be considered acceptable to give up some degree of realism in exchange for a more controlled setting. Moreover, the separation between the features of students and those of professionals is blurred and exhibits significant overlaps.
- We do not have enough findings to estimate how Client/Server Code Splitting may protect programs that are considerably different (e.g., larger or more complex) than the considered ones. This is one of the main directions for future work: including complexity metrics of the applications to protect as independent variables.
- Our subjects accessed the source code of the application to tamper with. This scenario does not correspond to the case of attackers that have only access to the binaries. However, since attacking binary code is more complex than attacking the source code, we considered ours as a worst case scenario. As a future work, we want to determine the variations in attack time and success rate when moving from sources to binaries.

[12] In the Italian University system, grades are assigned on a 30 values scale.

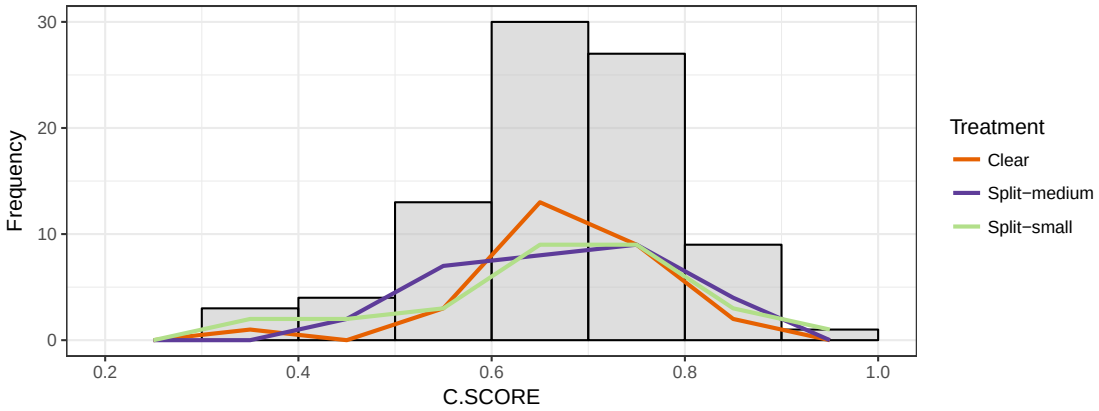



Fig. 8: Distribution of C skill among participants.

- Tools made available to subjects were up to date and valid representative of tools hackers may use. Even if hackers have several more types of tools at their disposal when they have to mount attacks, such additional tools are mostly used to obtain a usable representation of the program to attack (e.g., reversing, disassembling, decompiling tools). In our case, subjects had the source code. Therefore, using several of the more advanced hacking tools was pointless (i.e., IDE and debuggers were actually enough to mount the attack).
- The experiment made use of a single object, i.e., SpaceGame. We cannot be sure that applications with different structure or from a different domain would yield similar results, although the client-server structure of our application is fairly common and can be found in many systems.
- The object is small compared to several commercial programs that are targeted by hackers. Nevertheless, the asset we asked to compromise, i.e., functions and variables related to the movement of the space ship, as big as assets in larger applications. Our assumption is that while in real cases the time to locate the asset could be larger compared to our example, the time to modify the assets once located can be considered equivalent.

*Conclusion validity* threats affect the validity of the methods to derive outcomes from the treatment data.

- We have used non-parametric statistical methods and controlled the error rate (permutation test ANOVA, error rate corrected with the Bonferroni method) as presented in Section 3.10.
- We have collected data by means of survey questionnaires designed according to standard methods and scales [20].
- Tasks were similar and balanced (same attack task on the same application protected with different treatments); subjects were not heterogeneous, as they were all master students, and experiments avoided random irrelevance.
- The qualitative analysis of the reports relied on closed annotation (using an existing attack taxonomy) and on automated process mining (using the tool Disco), in order to minimize the risk of subjective qualitative inference. However, it is unavoidable that the same process diagrams might be interpreted differently by different researchers.

## 4 Results

The results are based on data collected during the experiment and made publicly available[13].

The population in the experiment includes a total of 87 participants. All participants are master students. The knowledge of C is roughly normal (Shapiro-Wilk test p=0.18) as shown in Fig. 8. We observe in the figure that no significant difference in C knowledge is observable among the participants assigned to the three treatment levels (MW p=0.87).

The general outcome from the experiments is summarized in Table 3.

Table 3: Summary statistics for Correctness and Elapsed Time.

| Treatment | n | n Correct | prop Correct | Time mean | Time sd |
|---|---|---|---|---|---|
| Clear | 28 | 25 | 0.89 | 93.71 | 25.93 |
| Split-medium | 30 | 20 | 0.67 | 93.07 | 29.19 |
| Split-small | 29 | 15 | 0.52 | 99.28 | 27.84 |

[13] https://github.com/torsec/splitting-experiments/tree/master/analysis

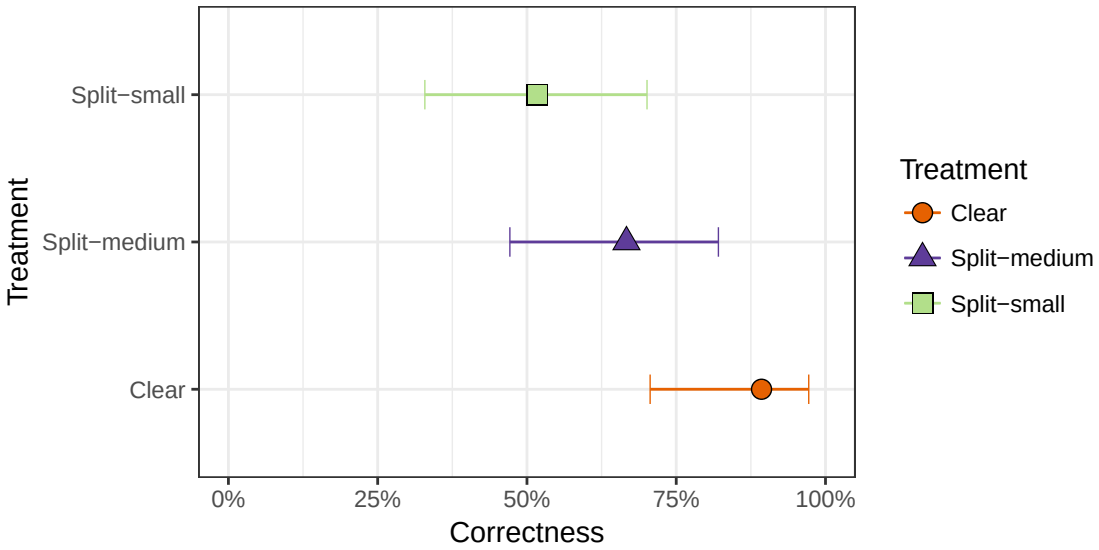



Fig. 9: Proportion of correctly completed tasks per Treatment with 95% confidence intervals.

Table 4: Logistic regression of Correctness vs. Treatment.

| | Estimate | Std. Error | z value | Pr(>\|z\|) |
|---|---|---|---|---|
| (Intercept) | 1.735 | 0.443 | 3.917 | 0.000 |
| TreatmentSplit-medium | -1.238 | 0.558 | -2.220 | **0.026** |
| TreatmentSplit-small | -1.666 | 0.578 | -2.881 | **0.004** |

Table 5: Permutation test Anova of Elapsed Time vs. Treatment, and Correctness.

| | Df | R Sum Sq | R Mean Sq | Iter | Pr(Prob) |
|---|---|---|---|---|---|
| Correct | 1 | 10038.641 | 10038.641 | 5000 | **<0.001** |
| Treatment | 2 | 310.655 | 155.327 | 78 | 0.641 |
| Correct:Treatment | 2 | 1784.008 | 892.004 | 746 | 0.354 |
| Residuals | 81 | 51396.272 | 634.522 | | |

### 4.1 Analysis of correctness

The proportion of correctly completed tasks for the three treatments is reported in Fig. 9. We observe different correctness levels based on the treatment and we checked the statistical significance of such differences by means of a logistic regression. The coefficient estimates and the relative significance level are reported in Table 4.

We observe that both splitting treatments (*Split-small* and *Split-medium*) have a significant effect on the correctness of the attack task results with respect to treatment *Clear* (the intercept). Looking at the coefficients, we observe that when treatment *Clear* is applied we have a very large odds ($e^{1.735} = 5.7$). The two split treatments, *Split-medium* ($e^{1.735-1.238} = 1.6$) and *Split-small* ($e^{1.735-1.666} = 1.1$), exhibit lower odds.

We computed the relative correctness odds ratios of the different pairs of treatment levels. The odds of a successful attack with Split-medium is 3.4 times lower than with clear code, while the odds are 5.3 times lower with Split-small. The reason that we conjecture for such a difference is the different number of sliced dependencies and correspondingly the different number of exchanged client-server messages (see Table 1). Hence, we can answer RQ1 as follows:

**RQ1 (correctness)**: *Code splitting reduces significantly the probability that attackers successfully complete code tampering. Small size code splitting is most effective, with the odds of successful attacks reduced by a factor 5.3. While being still significantly effective over clear code, with an odds ratio of 3.4, medium size code splitting is less effective than small size splitting, the reason being that medium size splitting cuts less data dependencies and requires less client-server messages than small size splitting.*

### 4.2 Analysis of time

We checked the effect of treatment and correctness on time by means of a permutation test Anova. Results are reported in Table 5. We observe a significant difference in terms of elapsed time between correct and wrong tasks. On the other

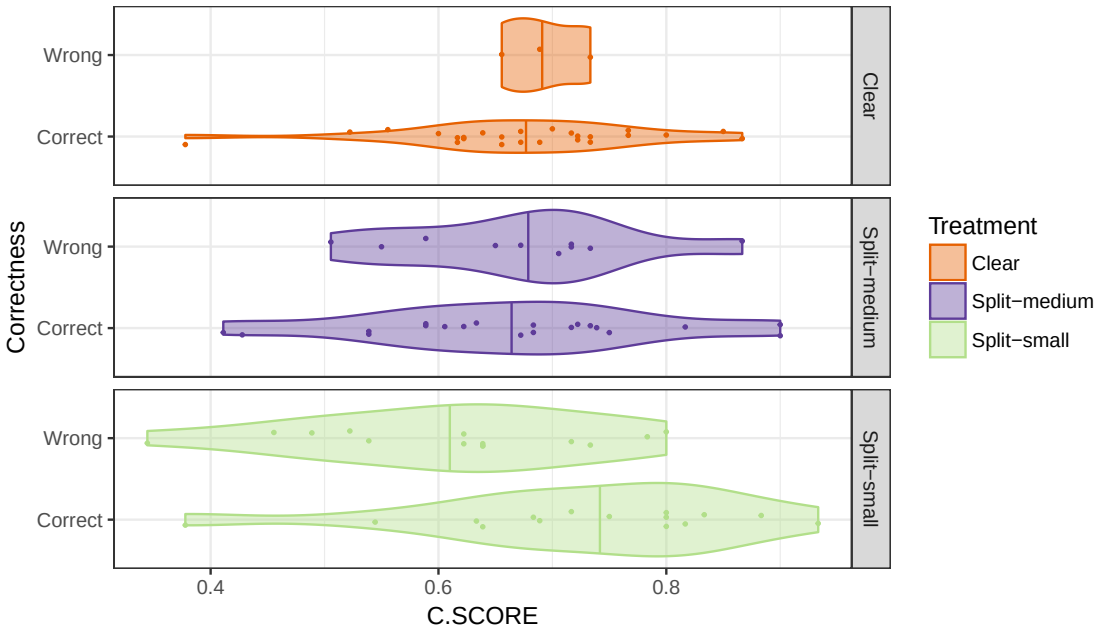



Fig. 10: Boxplot of time to complete the task.

hand the treatment appears to have no statistically significant effect on time; moreover no interaction between treatment and correctness was detected.

We can answer RQ2 as follows:

**RQ2 (time)**: *Since subjects tended to use the entire allotted time to complete and verify the results of code tampering, we could not observe any significant difference of the attack time, regardless of the protection applied to the code.*

### 4.3 Analysis of Co-factors

Since we verified that no significant difference exists in terms of C skill distribution among the different treatments (see Fig. 8), we can analyze the relationship of C.SCORE with correctness and time separately for each treatment. The significance level we consider in this analyses will be 1.71% (i.e. 5% / 3) according to the Bonferroni rule.

*C Skill and Correctness* The level of C Skill for different treatment and task outcome is reported in Fig. 10. Table 6 reports the coefficients and relative significance associated with a logistic regression of Correctness vs. C Skill score for the three distinct treatments.

Table 6: Logistic regression of Correctness vs. C Skill Score for different treatments.

| | Estimate | Std. Error | z value | Pr(>\|z\|) |
|---|---|---|---|---|
| Clear | | | | |
| (Intercept) | 3.406 | 4.618 | 0.738 | 0.461 |
| C.SCORE | -1.879 | 6.625 | -0.284 | 0.777 |
| Split-medium | | | | NA |
| (Intercept) | 1.021 | 2.212 | 0.461 | 0.645 |
| C.SCORE | -0.491 | 3.259 | -0.151 | 0.880 |
| Split-small | | | | |
| (Intercept) | -4.422 | 2.275 | -1.944 | 0.052 |
| C.SCORE | 6.682 | 3.311 | 2.018 | 0.044 |

We observe no significant effect of the C skill score on correctness. Notably for the *Split-small* treatment a large effect is detected, though not statistically significant. Probably this treatment was more challenging than the other two thus the skill played a more relevant role.

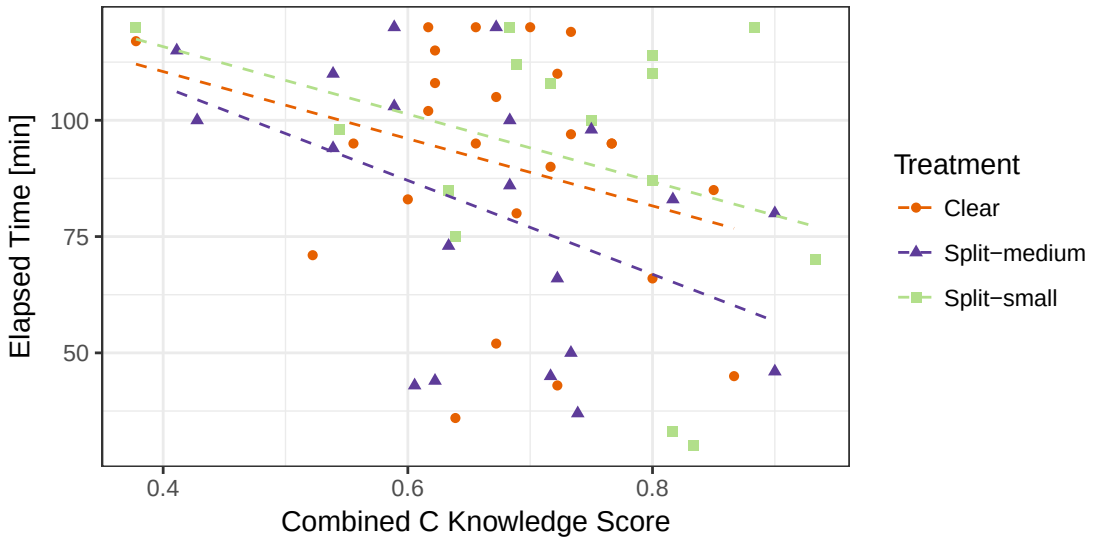



Fig. 11: Boxplot of time to complete the task vs. Treatment and C skill.

Table 7: Regression of Elapsed time vs. C Skill Score for different treatments.

| | Estimate | Std. Error | t value | Pr(>\|t\|) |
|---|---|---|---|---|
| Clear | | | | |
| (Intercept) | 139.335 | 33.656 | 4.140 | 0.000 |
| C.SCORE | -72.177 | 49.246 | -1.466 | 0.156 |
| Split-medium | | | | |
| (Intercept) | 147.574 | 30.365 | 4.860 | 0.000 |
| C.SCORE | -100.849 | 44.929 | -2.245 | 0.038 |
| Split-small | | | | |
| (Intercept) | 144.840 | 40.074 | 3.614 | 0.003 |
| C.SCORE | -72.532 | 54.204 | -1.338 | 0.204 |

*C Skill and Time* The relationship between elapsed time and C skill is reported in Fig. 11. This figure shows separate linear regressions for the three treatments. Table 7 reports the permutation test Anova of the regressions. The analysis shows that the C Skill score has no significant effect on the elapsed time for any treatment. The fitness of the fitted linear models are $R^2$ =0.05, $R^2$ =0.18, and $R^2$ =0.05 for the three treatments.

## 4.4 Analysis of the Attack Strategy

Table 8: Summary of attack strategy usage.

| Treatment | n | internal ( %) | external ( %) | mixed ( %) |
|---|---|---|---|---|
| Clear | 28 | 15 (53.6%) | 8 (28.6%) | 5 (17.9%) |
| Split-medium | 30 | 5 (16.7%) | 15 (50.0%) | 9 (30.0%) |
| Split-small | 29 | 8 (27.6%) | 15 (51.7%) | 5 (17.2%) |

The number and percentage of participants that applied the three attack types are reported in Table 8. Fig. 12 reports the success rate and the elapsed time for the participants who employed different strategies, divided by treatment. We note that 2 participants did not report enough information to let us infer which strategy was applied. We observe that depending on the type of treatment a specific approach was preferred to the others to attack the software. There is a statistically significant relationship between treatment and strategy category (p=0.043). Clear code was attacked most often with an internal approach. The odds of adopting external attacks are 4.9 times higher with Split-medium than with Clear code (p=0.013). The odds ratio is 2.5 with Split-small, though not statistically significant (p=0.11).

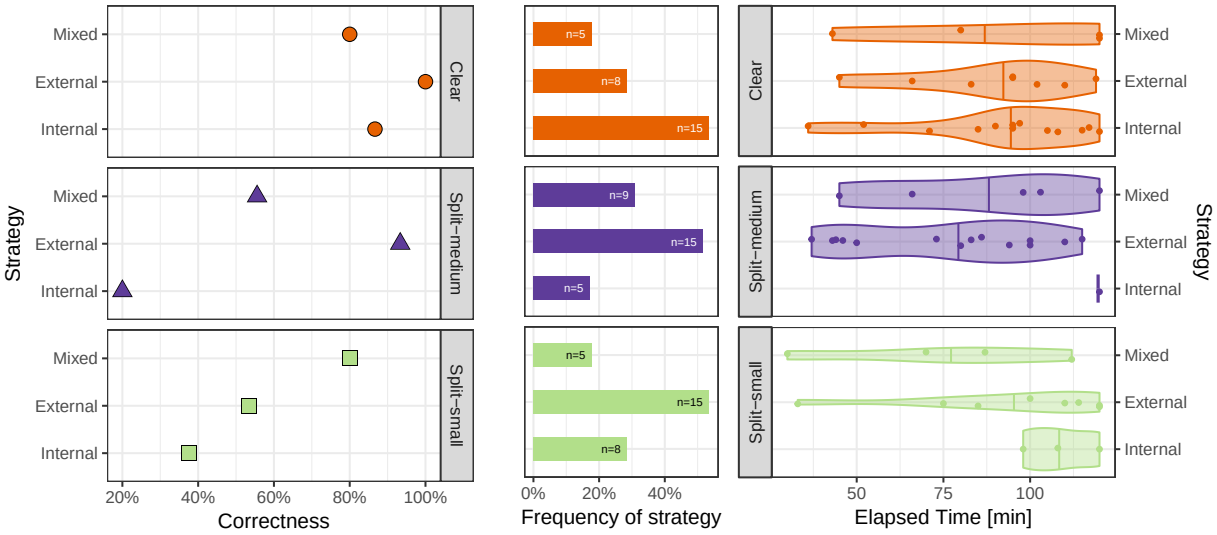



Fig. 12: Success rate, frequency, and time for different strategies by treatment.

Table 9: Logistic model regression coefficients and significance of success rate vs. attack strategy.

| | Estimate | Std. Error | z value | Pr(>\|z\|) |
|---|---|---|---|---|
| *Clear* | | | | |
| (Intercept) | 1.872 | 0.760 | 2.464 | **0.014** |
| External | 17.694 | 3802.118 | 0.005 | 0.996 |
| Mixed | -0.486 | 1.352 | -0.359 | 0.719 |
| *Split-medium* | | | | |
| (Intercept) | -1.386 | 1.118 | -1.240 | 0.215 |
| External | 4.025 | 1.524 | 2.642 | **0.008** |
| Mixed | 1.609 | 1.304 | 1.234 | 0.217 |
| *Split-small* | | | | |
| (Intercept) | -0.511 | 0.730 | -0.699 | 0.484 |
| External | 0.644 | 0.895 | 0.720 | 0.472 |
| Mixed | 1.897 | 1.335 | 1.421 | 0.155 |

### *4.4.1 Strategy effectiveness*

The influence of the strategy on the success rate can be best understood by means of a logistic model. The coefficients and the significance levels are reported in Table 9.

We observe a significant effect of the external strategy (p=0.008) with the *Split-medium* treatment. The odds of a successful attack with such a strategy are 14. Similarly, a significant effect of the internal strategy is observed for the *Clear* treatment, with odds of success 6.5.

Table 10: Model regression coefficients and significance for Time vs. Strategy.

| | Df | R Sum Sq | R Mean Sq | Iter | Pr(Prob) |
|---|---|---|---|---|---|
| Treatment | 2 | 82.25618 | 41.12809 | 53 | 1.0000000 |
| Strategy | 2 | 2463.71763 | 1231.85882 | 960 | 0.1427083 |
| Treatment:Strategy | 4 | 3138.58055 | 784.64514 | 404 | 0.6881188 |
| Residuals | 51 | 39031.78150 | 765.32905 | | |

The influence of the strategy on the time required to conduct a successful attack can be checked with a linear model. The coefficients and the significance levels of a permutation test are reported in Table 10. We observe no statistically significant effect of the strategy on the time elapsed in successful attacks and no interaction with the treatment.

**RQ3 (strategy)**: *The attack strategy affects the attack success rate significantly, while it does not affect the attack time. The odds of a successful attack are higher in the presence of code splitting when the attack strategy adopted is the external one, while on clear code it is the internal attack strategy that gives highest odds of success.*

Table 11: Concepts used to annotate the attack reports, ranked by occurrence frequency

| | | | |
|---|---|---|---|
| Tamper with the code statically | 395 | Identify assets by naming scheme | 9 |
| Analyze attack results | 355 | Identify protection | 8 |
| Understand code logic | 189 | Analysis / reverse engineering | 7 |
| Identify code containing sensitive assets | 174 | Black-box analysis | 7 |
| Preliminary understanding of the app | 85 | Dynamic analysis | 5 |
| Debugging | 51 | Identify output generation | 4 |
| String / name analysis | 51 | Make hypothesis on protection | 3 |
| Make hypothesis on reasons for attack failure | 49 | Data flow analysis | 2 |
| Build the attack strategy | 35 | Recognize anomalous/unexpected behaviour | 2 |
| Evaluate and select alternative steps / revise strategy | 30 | Assess effort | 1 |
| Identify input / data format | 27 | Build a workaround | 1 |
| Static analysis | 26 | Choose path of least resistance | 1 |
| Tamper with data | 22 | Evaluate attack results | 1 |
| Make hypothesis | 21 | Identify data format | 1 |
| Understand the app | 20 | Limit scope of attack | 1 |
| Identify API calls | 15 | Prepare attack | 1 |
| Identify points of attack | 15 | Prepare the environment | 1 |
| Confirm hypothesis | 11 | Tamper with code and execution | 1 |
| **Total** | | | **1934** |

### 4.5 Analysis of the Attack Process

The authors of this paper annotated all 87 attack reports that were collected from the participants. To reduce the subjectivity of the annotation process, each report was annotated twice, by two different authors. We adopted a closed annotation scheme, with the set of possible annotations consisting of the concepts available from the attack taxonomy described in a previous paper [9].

Table 11 lists the concepts used to annotate the reports, ordered by decreasing frequency of use as annotations. In total, 1934 annotations have been produced. The two most commonly used annotations, “Tamper with code statically” and “Analyze attack results” are quite representative of a typical iterative attack phase, consisting of repeated code modifications and analysis of the results achieved with such modifications. The next three most frequently used annotations are also representative of another important attack phase: code understanding and more specifically code understanding aimed at the identification of sensitive assets (concept “Identify code containing sensitive assets”). Such assets are the targets of the next attack phase.

Fig. 13 shows the attack process followed by all participants, regardless of the code under attack (either clear or protected) and regardless of the attack outcome (success or fail). The three activities conducted initially, “Preliminary understanding of the app”, “Identify input / data format” and “Understand code logic” deal with general program comprehension and represent a preliminary phase of the attack. The two core attack activities are “Tamper with code statically” and “Analyze attack results”. The code is modified to try to understand or circumvent a protection. The execution output after modifying the code is analyzed to obtain relevant information and to assess the effectiveness of the attack. Hence, this phase of the attack is inherently iterative. In fact, the process model in Fig. 13 includes several possibilities of loops in the central phase: from “Analyze attack results” back to “Tamper with code statically” passing either through “Understand code logic”, or through “Build the attack strategy”, or through “Identify code containing sensitive assets”, which is optionally followed by “String / name analysis”. The debugger is also used in this central phase (activity “Debugging”, followed by “Identify code containing sensitive assets” and then “Tamper with code statically”). While the activity “Build the attack strategy” is still part of the core iterative phase, “Make hypothesis on reasons for attack failure” is conducted mostly before terminating, probably unsuccessfully, the attack attempt (the lack of a backward link from “Make hypothesis on reasons for attack failure” to a previous activity does not mean that such link never occurred: it might have occurred infrequently, so that the process mining algorithm has ignored it).

Fig. 14 and 15 show the processes mined respectively from attack reports of successful vs. unsuccessful attacks. We can notice that the attack process followed in unsuccessful attempts is pretty simple (Fig. 15) and consists of a preliminary understanding phase followed by the iterative phase already described for the general process (Fig. 13) and involving a loop over three activities: (1) code understanding; (2) code modification; (3) attack result analysis. On the other hand, the process followed in successful attempts is more articulated and involves a more diversified set of activities. The overall structure is the same (see Fig. 14): a preliminary understanding phase followed by the iterative attack phase, but in both phases it is possible to identify specific activities that play an important role, while being quite irrelevant in the unsuccessful attack process. Among them, “Build the attack strategy” is quite meaningful, since it indicates a planning phase that is not apparent in the unsuccessful attack process. A more focused understanding activity, targeting the identification of code containing sensitive assets (“Identify code containing sensitive assets”), is another major component of the successful attack process which does not play any major role in the unsuccessful process.

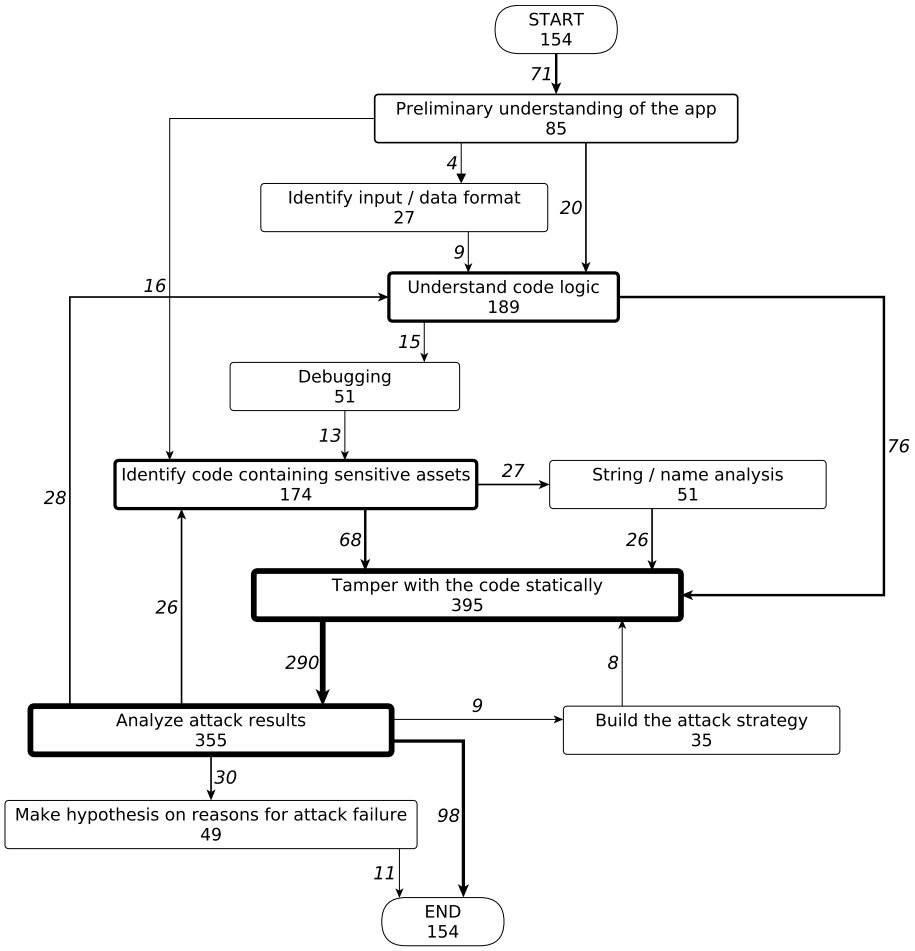



Fig. 13: Attack process mined from all annotated reports

Fig. 16 and Fig. 17a and 17b show the attack process followed by participants who worked respectively on clear code, on code protected by Split-medium and by Split-small. The most remarkable difference between the first process and the other two processes is the iterative nature of the central attack phase. While participants attacking clear code performed several iterations in the central attack phase, between "Tamper with code statically" and "Analyze attack results", passing through other activities, such as "Make hypothesis on reasons for attack failure" and "Identify code containing sensitive assets", participants attacking protected code (both with medium and small-size splitting) adhered mostly to a waterfall, strictly phased process, where substantial effort is first devoted to one activity, then to the next one, and so on, with little room for iterations (as remarked above, the lack of backward links does not mean that they never occurred: they might have occurred infrequently). This seems to indicate that each attack step against protected code is very effort intensive and requires substantial time allocation. As a consequence, the possibility to iterate the attack phase is quite limited.

**RQ4 (process)**: *The attack process includes an initial understanding phase, followed by an iterative phase, in which code tampering and analysis of attack results are repeatedly executed. In successful attacks, the process is more articulated and it includes a phase devoted to planning and strategy building. When the code is protected by code splitting, attackers are constrained to a phased, waterfall process, whose steps involve substantial effort, leaving little room for an iterative approach.*

## 5 Discussion

Here we report the implications and general observations that we can formulate, based on the objective and quantitative results presented in the previous section.

**Correctness.** *The amount of dependencies cut by a slice is more important than the criterion size.* The two splitting configurations have been named after the size of their criteria, with a smaller criterion corresponding to Split-small and a larger criterion to Split-medium. Based on our intuition, we expected Split-medium to be more complex, hence more difficult to attack, than Split-small. However, we observed the opposite trend: the success rate was found empirically to be higher on Split-medium than on Split-small (see Fig. 9 and Table 4).

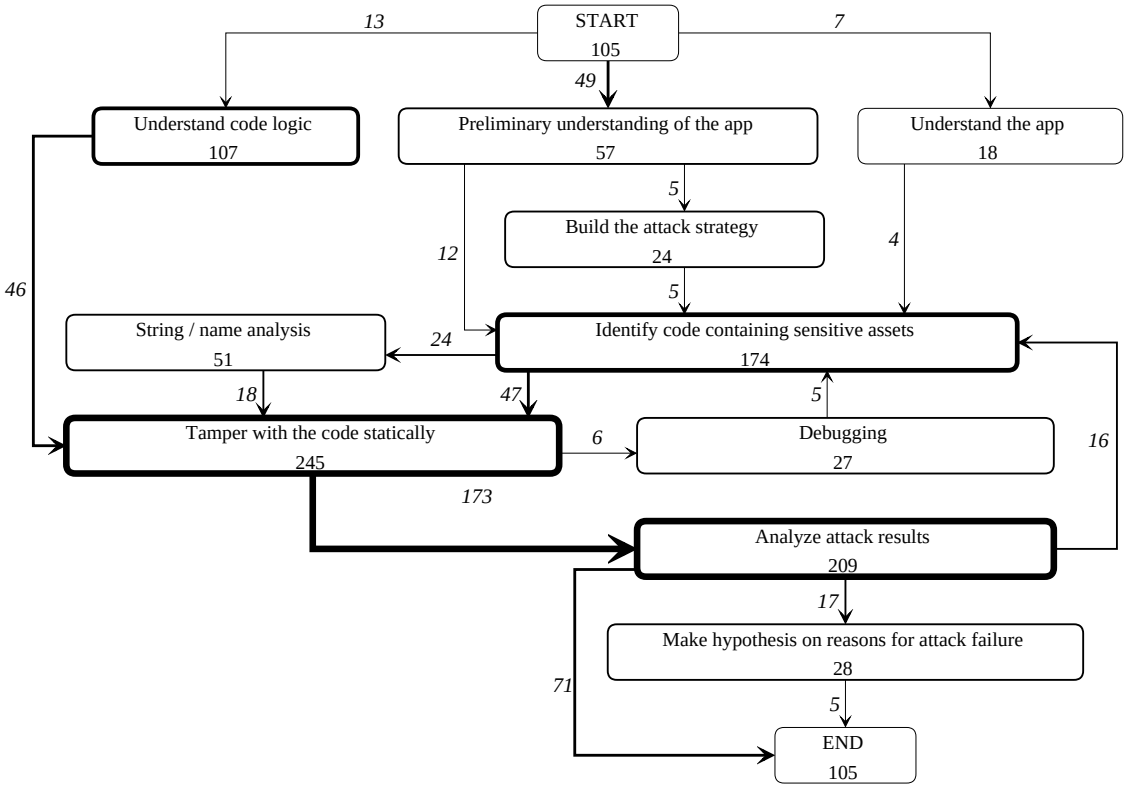



Fig. 14: Attack process mined from annotated reports of successful attacks

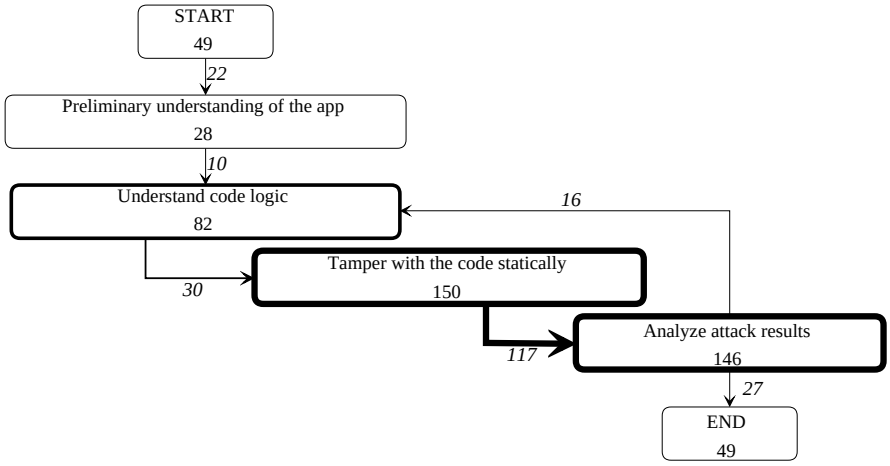



Fig. 15: Attack process mined from annotated reports of unsuccessful attacks

The reason for the lower success rate on Split-small is probably the higher number of dependencies cut by this splitting configuration, causing the slice to be larger and correspondingly requiring a larger number of messages exchanged with the split server. Thus, to elaborate an effective protection configuration a developer should keep in mind that the size of annotated code could be misleading. A better metric to consider when applying splitting is the final slice size or the number of dependencies it involves.

**Time.** *The entire time granted to attackers was spent on the task.* The time used to complete a successful attack is similar across the three techniques and quite close to the maximum time available in the experimental session (see Fig. 18). The reason is that even when a participant realized that the task was complete, she/he spent the remaining

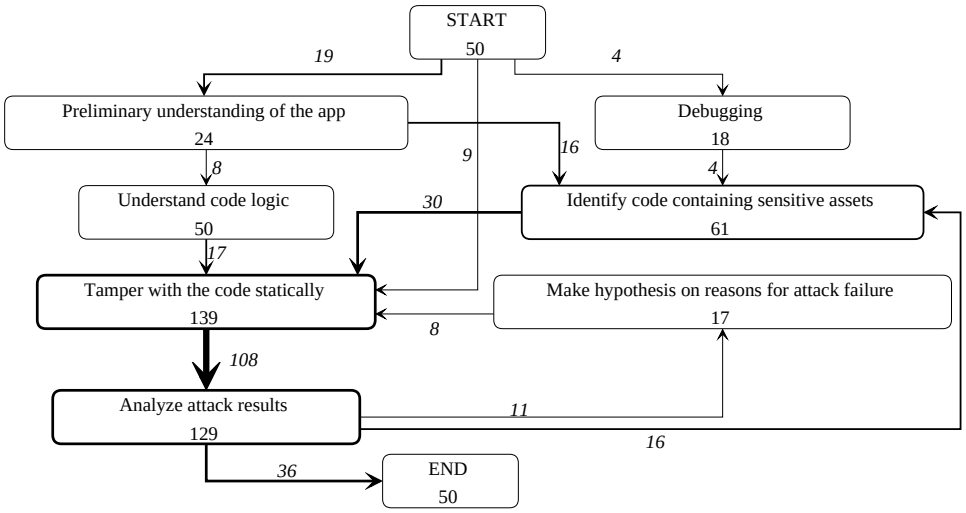



Fig. 16: Attack process mined from annotated reports of attackers working on clear code

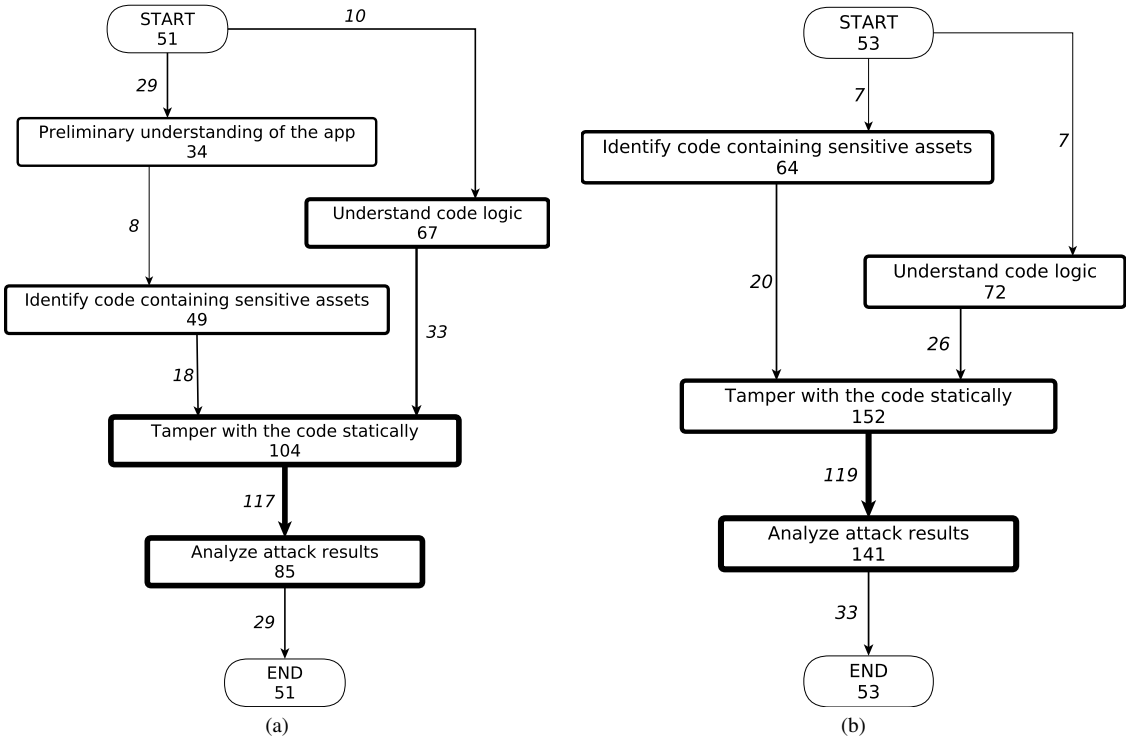



Fig. 17: Attack processes mined from annotated reports of attackers working on code protected by medium-size (a) and small-size (b) splitting

time to validate the solution, or to work on collateral aspects that have marginal importance for the final objective. For instance, some participants reported to have spent the remaining time to fix rarely occurring program crashes, not strictly related to the attack task.

We think that this behaviour is peculiar of the artificial experimental setting of a controlled experiment conducted in a classroom. In a public challenge, an attacker is willing to deliver a working solution as soon as possible. In fact, in a public challenge, only the first solution wins the bounty. Hence, it makes sense to rush and post a badly implemented but working attack.

**Attacker skill.** *Advanced skills seem to be important when the attack task is hard.* Participants' skill is not playing a significant role neither on the correctness nor on the time of the attack according to the $p$-values, which are never significant if we apply a threshold $\alpha = 0.05$ and the Bonferroni correction. The $p$-value gets very near to significance ($p$-value = 0.044) as the task becomes harder to complete (i.e., in the case of Split-small; see Table 6).

Indeed, Fig. 11 shows a clear decreasing trend of the attack time as the C knowledge score increases. Lack of statistical significance of such trend might be due to low statistical power: the number of data points available might be insufficient to discriminate multiple cofactors (correctness/time, treatment and skills) in a statistically significant way.

**Attack strategy**. *External approaches are effective on protected code; mixed approaches are more effective on hard tasks.* Depending on the problem at hand, a different attack strategy should be adopted to maximize the chances of success (see Fig. 12 and Table 9). A single, simple strategy is more effective when the code is not protected (internal strategy on clear code) or when a lightweight protection is deployed (external strategy on Split-medium). However, when the code is protected with a stronger approach, a more complex and adaptive strategy is needed (mixed strategy on Split-small).

A strategy that involves multiple (external and internal) view points is probably more effective because it leads to a wider understanding of the portion of code to attack and it helps the attacker deliver a working solution faster (this trend is evident in Fig. 12, even if not supported by statistical significance). Participants who considered both perspectives (external and internal) when understanding the code were more successful than those who considered just one perspective, probably because they acquired better knowledge of the code and of the protection. The mixed strategy might be also an indicator of higher flexibility in code exploration, which in turn is a factor that can contribute to successful attacks.

**Attack process.** *Succesfull attacks require strategic thinking encompassing a planning phase that considers multiple strategies.* The above considerations at the strategy level are confirmed by analysis of the process adopted in successful attacks (see Fig. 14 and Section 4.5). Successful attacks involve complex, elaborated attack processes, including phases such as attack planning and focused understanding. In these phases, different approaches and strategies are considered, compared, evaluated and (possibly) composed into a process that mixes more attack approaches. Moreover, the presence of code protection reduces the possibility of iterating the attack activities several times, since each activity in isolation becomes much more time consuming and effort demanding. The short time available for the attack does not allow for many iterations. Correspondingly, a complex attack strategy becomes more effective. In fact, processes based on a complex attack strategy requires the attacker to reason from multiple perspectives – e.g. internal/external – concurrently.

**Limitations.** We ought to remark that the above findings, although stated in assertive form, must be read as in conditional form. Our results are based on a single controlled experiment that is based on a single object, with student participants performing a single task. We strived to make the context of our experiment as clear as possible and reported a full discussion of the threats to the validity of our results in Section 3.11. This study represents a piece of evidence that needs to be confirmed – possibly corrected, or altogether refuted – by further future studies.

## 6 Related Work

In the literature, two main approaches are used for the assessment of software protection techniques: (1) theoretical evaluation based on software metrics; (2) empirical evaluation, based on controlled experiments involving students or on case studies with professional hackers. In this work we focus on assessing Client/Server Code Splitting with a controlled experiment involving students as participants.

**Assessment based on code metrics.** One of the early examples of the first, metrics-based approach is represented by *potency*. Collberg et al. introduced this concept as a metric to estimate the effectiveness of obfuscation transformations on programs [11]. Relying on the potency definition, Anckaert et al. presented a comparison of obfuscation techniques [1]. Another evaluation based on software metrics is given by Linn et al.. They considered the confusion factor as a measurement, which estimates the number of binary instructions that a code decompiler is not able to parse [19]. Goto et al. presented a methodology to measure obfuscation transformations in terms of complexity based on the compiler syntax analysis [16]. Visaggio et al. instead exploited code entropy as a transformation effectiveness metric for obfuscated Javascript code [23]. Ceccato et al. evaluated the impact of several obfuscation techniques on Java code quality [4]. The authors performed a large set of experiments and estimated the effects of obfuscation on ten different complexity and modularity metrics. To the best of our knowledge, the most investigated software protection technique in the literature is obfuscation, while other protection techniques (such as Client/Server Code Splitting) were not assessed in the terms we presented so far. Recently, alternatives to fully theoretical metrics based evaluations have been proposed. Canavese et al. presented a method to estimate *a priori* the effectiveness of software obfuscation by means of artificial neural networks [3].

**Assessment based on controlled experiments with students.** On the other hand, evaluations by means of experiments with human subjects were introduced by Sutherland et al., who presented the first study in this field [22]. In such study they report direct correlation between correctness of reverse engineering tasks and attackers expertise. In addition, they provided evidence about the limitations of source code metrics in estimating the attack delay when binary code is involved. This work highlighted the worth and importance of empirical assessment. Ceccato et al. measured the correctness and effectiveness in understanding and modifying decompiled obfuscated Java code, compared to decompiled clear code [8]. The measurement involved two controlled experiments. In a successive work, the same authors presented an extension with a larger set of experiments conducted on several obfuscation techniques [7]. Then, their replication package has been used by Hänsch et al. to conduct a similar experiment and assess a slightly different set of obfuscations [17]. Viticchié et al. empirically evaluated the attack delay introduced by a data obfuscation technique, namely

*variable merge*. They confirmed that attacks are still possible on protected programs, but they are delayed by a factor of six for that technique [25].

The major novelty of the present empirical study with respect to previous empirical works is the obfuscation considered in the study. In fact, previous works studied only code obfuscations, e.g., hiding the program's control flow and variable names. In the present work, we focus on a transformation, Client/Server Code Splitting, that removes critical assets from the delivered code with the purpose of protection. While the general findings of previous works were confirmed, specific findings on the attack strategy and process obtained in the present study are specific of the considered code protection technique.

**Assessment based on experiments with professional participants.** Ceccato et al. presented one of the first empirical assessments of protection techniques from the perspective of professional hackers [9]. The authors conducted an experiment in which fully protected applications (i.e., multiple protections were applied to simulate real world protected applications) have been subjected to tampering attacks by expert white hackers, to assess how protection is perceived and approached by professional attackers. The same authors presented a follow-up experiment, where a public challenge has been launched (with a cash bounty) to confirm the pieces of evidence collected in the previous study in a broader context, with professional white hat hackers [10].

Involving expert hackers would have allowed the experimental context to be more similar to real attack settings, but at the cost of limiting the observability of the phenomenon. In fact, as reported by Ceccato et al. [9, 10], it is not possible to involve expert hacker teams in controlled environments, where precise measurements (e.g., of time) and specific experimental constraints are applied. Moreover, while professional hackers usually work in small teams, with a larger group of students statistical tests can be applied to study significance.

Our work extends the assessment of the effectiveness of protection techniques by means of empirical experiments, by addressing a technique, Client/Server Code Splitting, which was never assessed before. Since such technique has specific traits that makes it substantially different from code and data obfuscation, the present study represents the first one, to the best of out knowledge, that sheds light on the attack strategy and process followed when an application is protected by Client/Server Code Splitting, as well as on the strength of the protection in relation to the slicing criteria used for code splitting.

## 7 Conclusion

We have reported and analysed the data collected in a controlled experiment on the effectiveness of Client/Server Code Splitting as a software protection technique. The experimental subjects were 87 MSc students who executed an attack task either on clear or on protected C code. The task assigned to subjects simulated the situation in which a game player aims at gaining an unfair advantage over the other players by tampering with the game software.

Experimental results show that code splitting introduces a substantial barrier to code tampering, by reducing the chances of a successful attack by (up to) 5.3 times. Among the factors that affect the subjects' capability of mounting a successful attack, the most important ones are the attack strategy and process. Especially when "hard to break protections" are considered, successful attacks are associated with a mixed attack strategy that iterates through internal and external code manipulations to gain knowledge on the protections and to validate the initial attack attempts, which are usually unsuccessful. Correspondingly, the attack process is also articulated and involves code understanding, hypothesis making, attack planning, code tampering and analysis of attack results. On the other hand, we found that programming skills play a relevant role only when the protection is particularly complex.

Our results are relevant for protection developers and users. In fact, we found that the amount of dependencies cut by a slicing criterion is more important than its size in determining the effectiveness of the protection. An easy way to estimate such dependencies consists of looking at the number of messages exchanged between client and server, since each cut dependency will require at least one client-server communication. Moreover, we noticed that the attack process involves substantial code understanding, which is not explicitly obstructed by Client/Server Code Splitting. This indicates that the combination of Client/Server Code Splitting with code obfuscation techniques might amplify the effects of the protection. It would be interesting to design and conduct an empirical study where such combined effects are quantified.

In addition to such study, in our future work we plan to extend the validity of the present study along the following directions: (1) considering programs to be attacked from domains different from gaming (e.g., mobile applications); (2) including reverse engineering from the binaries as an attack step, to analyse how it interacts with the other attack steps investigated in the present paper; (3) try to involve professionals instead of students, better if in controlled experiments. We strongly believe that replication is a cornerstone of our discipline, therefore all the material necessary to replicate and extend our study has been made publicly available[14].

## References

1. Anckaert, B., Madou, M., De Sutter, B., De Bus, B., De Bosschere, K., Preneel, B.: Program obfuscation: a quantitative approach. In: Proc. ACM Workshop on Quality of protection, pp. 15–20 (2007)

[14] https://github.com/torsec/splitting-experiments

2. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K.: On the (im) possibility of obfuscating programs. Lecture Notes in Computer Science **2139**, 19–23 (2001)
3. Canavese, D., Regano, L., Basile, C., Viticchié, A.: Estimating software obfuscation potency with artificial neural networks. In: International Workshop on Security and Trust Management, pp. 193–202. Springer (2017)
4. Ceccato, M., Capiluppi, A., Falcarin, P., Boldyreff, C.: A large study on the effect of code obfuscation on the quality of java code. Empirical Software Engineering **20**(6), 1486–1524 (2015)
5. Ceccato, M., Dalla Preda, M., Nagra, J., Collberg, C., Tonella, P.: Trading-off security and performance in barrier slicing for remote software entrusting. Automated Software Engineering **16**(2), 235–261 (2009)
6. Ceccato, M., Dalla Preda, M., Nagra, J., Collberg, C.S., Tonella, P.: Barrier slicing for remote software trusting. In: SCAM, pp. 27–36 (2007)
7. Ceccato, M., Di Penta, M., Falcarin, P., Ricca, F., Torchiano, M., Tonella, P.: A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. Empirical Software Engineering **19**(4), 1040–1074 (2014)
8. Ceccato, M., Di Penta, M., Nagra, J., Falcarin, P., Ricca, F., Torchiano, M., Tonella, P.: The effectiveness of source code obfuscation: An experimental assessment. In: IEEE 17th International Conference on Program Comprehension (ICPC), pp. 178–187 (2009). DOI 10.1109/ICPC.2009.5090041
9. Ceccato, M., Tonella, P., Basile, C., Coppens, B., De Sutter, B., Falcarin, P., Torchiano, M.: How professional hackers understand protected code while performing attack tasks. In: Proceedings of the 25th IEEE International Conference on Program Comprehension (ICPC) (2017)
10. Ceccato, M., Tonella, P., Basile, C., Falcarin, P., Torchiano, M., Coppens, B., De Sutter, B.: Understanding the behaviour of hackers while performing attack tasks in a professional setting and in a public challenge. Empirical Software Engineering (EMSE) pp. 1–47 (2018)
11. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Technical Report 148, Dept. of Computer Science, The Univ. of Auckland (1997)
12. Collberg, C., Thomborson, C., Low, D.: Watermarking, tamper-proofing, and obfuscation - tools for software protection. IEEE Transactions on Software Engineering **28** (2002)
13. Falcarin, P., Collberg, C., Atallah, M., Jakubowski, M.: Guest editors' introduction: Software protection. IEEE Software **28**(2), 24–27 (2011). DOI 10.1109/MS.2011.34
14. Feldt, R., Zimmermann, T., Bergersen, G.R., Falessi, D., Jedlitschka, A., Juristo, N., Münch, J., Oivo, M., Runeson, P., Shepperd, M., Sjøberg, D.I.K., Turhan, B.: Four commentaries on the use of students and professionals in empirical software engineering experiments. Empirical Software Engineering **23**(6), 3801–3820 (2018)
15. Futcher, L., von Solms, R.: Guidelines for secure software development. In: Proceedings of SAICSIT, pp. 56–65. ACM, New York, NY, USA (2008)
16. Goto, H., Mambo, M., Matsumura, K., Shizuya, H.: An approach to the objective and quantitative evaluation of tamper-resistant software. In: Third Int. Workshop on Information Security, pp. 82–96. Springer (2000)
17. Hänsch, N., Schankin, A., Protsenko, M., Freiling, F., Benenson, Z.: Programming experience might not help in comprehending obfuscated source code efficiently. In: Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018), pp. 341–356 (2018)
18. Krinke, J.: Barrier slicing and chopping. In: SCAM, pp. 81–87 (2003)
19. Linn, C., Debray, S.: Obfuscation of executable code to improve resistance to static disassembly. In: Proc. ACM Conf.Computer and Communications Security, pp. 290–299 (2003)
20. Oppenheim, A.N.: Questionnaire Design, Interviewing and Attitude Measurement. Pinter, London (1992)
21. R Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2016). URL `https://www.R-project.org/`
22. Sutherland, I., Kalb, G.E., Blyth, A., Mulley, G.: An empirical examination of the reverse engineering process for binary files. Computers & Security **25**(3), 221–228 (2006)
23. Visaggio, C.A., Pagin, G.A., Canfora, G.: An empirical study of metric-based methods to detect obfuscated code. International Journal of Security & Its Applications **7**(2) (2013)
24. Viticchié, A., Basile, C., Avancini, A., Ceccato, M., Abrath, B., Coppens, B.: Reactive attestation: Automatic detection and reaction to software tampering attacks. In: Proceedings of the 2016 ACM Workshop on Software PROtection, pp. 73–84. ACM (2016)
25. Viticchié, A., Regano, L., Torchiano, M., Basile, C., Ceccato, M., Tonella, P., Tiella, R.: Assessment of source code obfuscation techniques. In: Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on, pp. 11–20. IEEE (2016)
26. Weiser, M.: Program slicing. In: Proceedings of the 5th International Conference on Software Engineering, ICSE '81, pp. 439–449. IEEE Press, Piscataway, NJ, USA (1981). URL `http://dl.acm.org/citation.cfm?id=800078.802557`
27. Wheeler, B., Torchiano, M.: lmPerm: Permutation tests for linear models (2016). URL `http://CRAN.R-project.org/package=lmPerm`. R package version 2.1.0
28. Wohlin, C., Runeson, P., Hst, M., Ohlsson, M.C., Regnell, B., Wessln, A.: Experimentation in Software Engineering. Springer Publishing Company, Incorporated (2012)

**Author Biography**

**Alessio Viticchié** received the M.Sc. degree in Computer Engineering in 2015 from the Politecnico di Torino, where he is currently a research assistant and a Ph.D. student. His research interests are concerned with software security, software attestation, and software protection techniques design and assessment.

**Leonardo Regano** is a Ph.D student and research assistant at Politecnico di Torino, where he received the M.Sc. degree in Computer Engineering in 2015. His current research interests focus on software security, applications of artificial intelligence and machine learning to cybersecurity, analysis of security policies, and assessment of software protection techniques.

**Cataldo Basile** received a M.Sc. (summa cum laude) in 2001 and a Ph.D. in Computer Engineering in 2005 from the Politecnico di Torino, where is currently a research associate. His research is concerned with policy-based management of security in networked environments, policy refinement, general models for detection, resolution and reconciliation of specification conflicts, and software security.

**Marco Torchiano** is an associate professor at the Control and Computer Engineering Dept. of Politecnico di Torino, Italy; he has been post-doctoral research fellow at Norwegian University of Science and Technology (NTNU), Norway. He received an MSc and a PhD in Computer Engineering from Politecnico di Torino. He is Senior Member of the IEEE and member of the software engineering committee of UNINFO (part of ISO/IEC JTC 1). He is author or co-author of over 140 research papers published in international journals and conferences, of the book 'Software Development—Case studies in Java' from Addison-Wesley, and co-editor of the book 'Developing Services for the Wireless Internet' from Springer. He recently was a visiting professor at Polytechnique Montréal studying software energy consumption. His current research interests are: green software, UI testing methods, open-data quality, and software modeling notations. The methodological approach he adopts is that of empirical software engineering.

**Mariano Ceccato** is tenured researcher in FBK (Fondazione Bruno Kessler) in Trento, Italy. He received the PhD in Computer Science from the University of Trento in 2006 with the thesis "Migrating Object Oriented code to Aspect Oriented Programming". His research interests include security testing, empirical software engineering, code hardening, reverse engineering and re-engineering. He was program co-chair of the 12th IEEE Working Conference of Source Code Analysis and Manipulation (SCAM 2012), held in Riva del Garda, Italy.

**Paolo Tonella** is Full Professor at the Faculty of Informatics and at the Software Institute of Università della Svizzera Italiana (USI) in Lugano, Switzerland. He is also Honorary Professor at University College London, UK. Until mid 2018 he has been Head of Software Engineering at Fondazione Bruno Kessler, Trento, Italy. Paolo Tonella holds an ERC Advanced grant as Principal Investigator of the project PRECRIME. He received his PhD degree in Software Engineering from the University of Padova, in 1999, with the thesis "Code Analysis in Support to Software Maintenance". In 2011 he was awarded the ICSE 2001 MIP (Most Influential Paper) award, for his paper: "Analysis and Testing of Web Applications". He is the author of "Reverse Engineering of Object Oriented Code", Springer, 2005, and of "Evolutionary Testing of Classes", ISSTA 2004. Paolo Tonella was Program Chair of ICSM 2011 and ICPC 2007; General Chair of ISSTA 2010 and ICSM 2012. He is/was associate editor of TOSEM/TSE and he is in the editorial board of EMSE and JSEP. His current research interests include code analysis, web testing, search based test case generation and the test oracle problem.

## Additional analyses

Two additional analyses are reported for the sake of completeness, though they are not central for the main findings of this work. The first analysis concerns the distribution of the time spent to complete the task and the second reports the correlation between attack task correctness and the self-claimed experience with the C language, optionally expressed by a subset of the subjects.

### Elapsed time

The distribution of elapsed time required to complete the task is reported in Fig. 18 for both successful and failed attacks. By looking at Fig. 18, we can see that the tasks with a wrong solution are all very high and compressed against the 120 min time limitation. Such phenomenon is probably due to the participant trying to use all the allotted time to attack the code without succeeding.

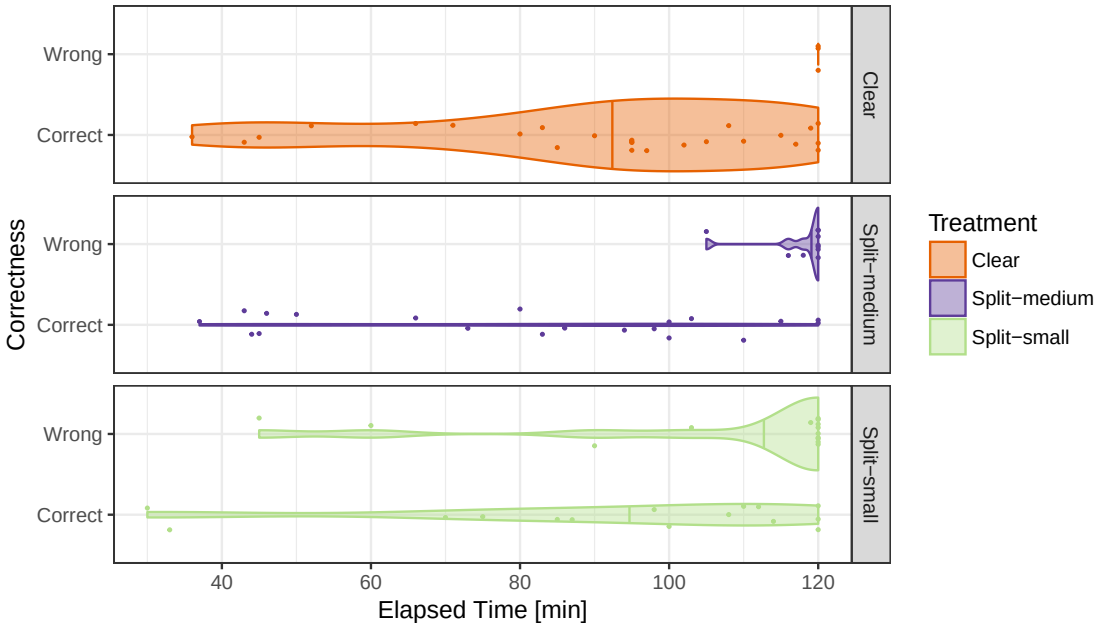



Fig. 18: Boxplot of time to complete the task.

### Experience of participants with the C language

Since the questionnaire about personal experience was not mandatory for the participants, we were able to collect information concerning only 35 students (out of 87).

Fig. 19 reports the average correctness (with the relative 95% confidence interval) for the subjects reporting different levels of experience with the C language. We cannot observe any clear and statistically significant trend.

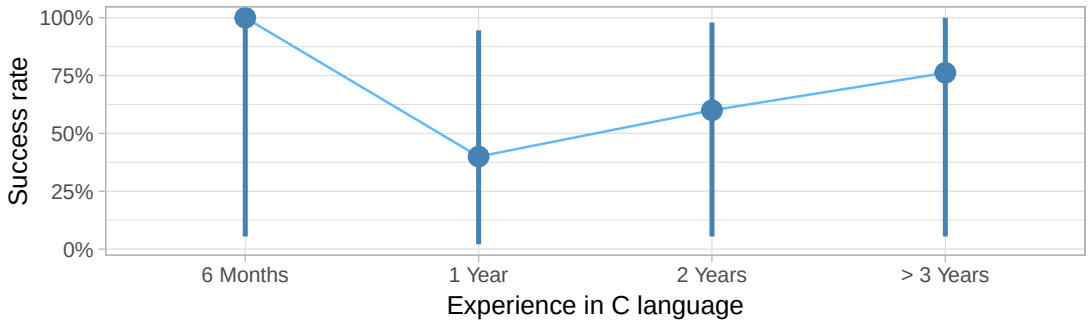



Fig. 19: Experience vs. Overall Correcteness.

A more detailed view can be observed by looking at the success rate when controlling for the treatment. Fig.20 reports the average correctness by experience and treatment. Also in this case the effect of experience is difficult to identify.

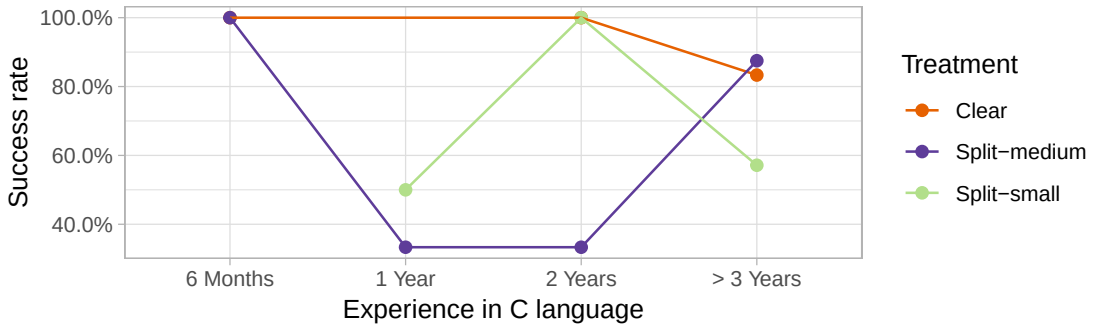



Fig. 20: Correcteness vs. Experience by Treatment.

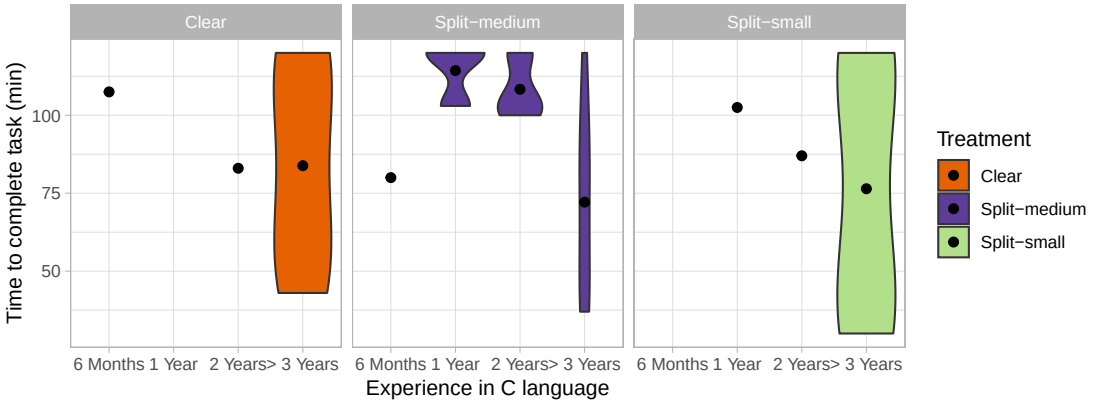



Fig. 21: Experience vs. Correcteness by Treatment.

Fig. 21 reports the effect of experience on the time to complete the attack task, divided by treatment. We observe a general reduction in time as experience increases though such effect is not statistically significant (too few data points and too much variability).

Finally, we report the relationship between the experience and the strategy selected, divided by treatment. This is reported as a heat map in Fig. 22. Also in this case we do not observe any clear effect of experience on the strategy selection.

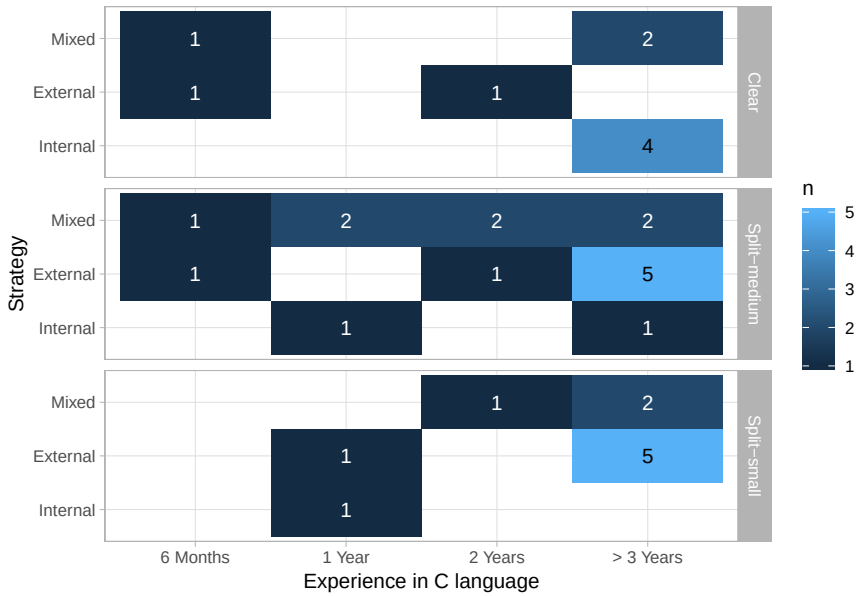



Fig. 22: Experience vs. Strategy by Treatment.

## Call graph of the SpaceGame application

We report in Fig. 23 for the sake of completeness, the call graph of the object of our experiment, the SpaceGame program, extracted with an ad hoc java program based on antlr [15] and JGraphT[16].

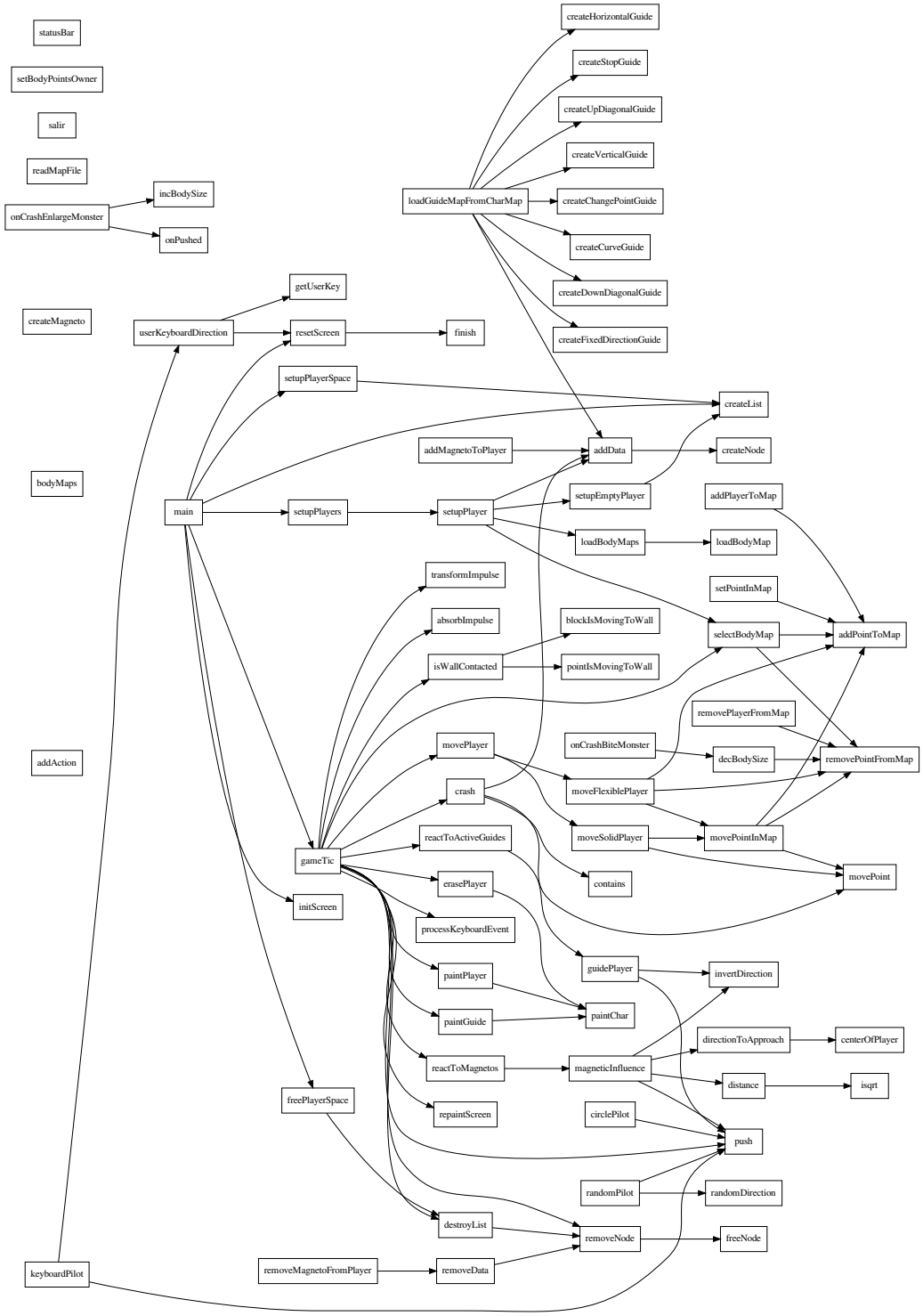



Fig. 23: Call graph of the clear version of the SpaceGame program.

[15] https://www.antlr.org/

[16] https://jgrapht.org/