Computers & Security 166 (2026) 104881

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Full Length Article

Empirical assessment of the code comprehension effort needed to attack programs protected with obfuscation

Leonardo Regano [a,*], Daniele Canavese [b], Cataldo Basile [c], Marco Torchiano [c]

[a] Dipartimento di Ingegneria Elettrica e Elettronica, Università di Cagliari, Via Marengo 3, 09123, Cagliari, Italy
[b] Istituto di Matematica Applicata e Tecnologie Informatiche "E. Magenes" (IMATI), Consiglio Nazionale delle Ricerche, Via de Marini, 6, 16149, Genova, Italy
[c] Dipartimento di Automatica e Informatica, Politecnico di Torino, Corso Duca degli Abruzzi, 24, 10129, Torino, Italy

ARTICLE INFO

Keywords:
Obfuscation
Man-at-the-end attacks
Reverse engineering
Control flow flattening
Opaque predicates
Attacker effort
Empirical assessment

ABSTRACT

Evaluating the effectiveness of software protection is crucial for selecting the most effective methods to safeguard assets within software applications. Obfuscation involves techniques that deliberately modify software to make it more challenging to understand and reverse-engineer, while maintaining its original functionality. Although obfuscation is widely adopted, its effectiveness remains largely unexplored and not rigorously evaluated. This paper presents a controlled experiment involving Master's students performing code comprehension tasks on applications hardened with obfuscation. The experiment's goals are to assess the effectiveness of obfuscation in delaying code comprehension by attackers and to determine whether complexity metrics can accurately predict the impact of these protections on success rates and durations of code comprehension tasks. The study is the first to evaluate the effect of layering multiple obfuscation techniques on a single piece of protected code. It also provides experimental evidence of the correlation between objective metrics of the attacked code and the likelihood of a successful attack, bridging the gap between objective and subjective approaches to estimating potency. Finally, the paper highlights significant aspects that warrant additional analysis and opens new avenues for further experiments.

1. Introduction

Software is routinely threatened by various attackers who seek to misuse applications, steal intellectual property, or use software as a vector for more extensive attacks, such as malware infections. This attack scenario is referred to as a "Man-At-The-End (MATE) attack" (Falcarin et al., 2011). MATE attacks can be carried out on the attacker's machine, where an array of tools is available to reverse engineer and tamper with the target applications. These tools include static, dynamic, symbolic, and concolic analysis tools, as well as debuggers, disassemblers, and decompilers, among others.

Hence, to protect their assets, software developers must rely on software protection techniques deployed directly into the software application and on remote, trusted servers under their control. Among these techniques, obfuscation plays a major role. Obfuscation encompasses a family of techniques that intentionally alter the target application's code to make it more difficult to understand, analyse, or reverse-engineer, while preserving the application's functionality (Collberg et al., 1997). Although obfuscation cannot, in theory, prevent MATE attacks (Barak et al., 2001), in practice it is highly effective at delaying adversaries (Viticchié et al., 2016b). This protection significantly increases the difficulty and cost of reverse-engineering code, thereby acting as a deterrent. In real-world scenarios, an attack may be deemed unsuccessful when goals are not achieved within a reasonable timeframe, either because the expected reward no longer justifies the extended effort or because a new software version is released, forcing attackers to (almost) start over (Basile et al., 2023).

Commercial and open-source tools are available to automatically apply obfuscation techniques, protecting target applications. However, the entire software protection field faces severe challenges. Commercial software protection tools and consulting services are expensive and opaque, relying heavily on *security-through-obscurity*. There is no generally accepted method for evaluating the effectiveness of software protection (Basile et al., 2023).

In this context, determining the impact of obfuscation is a crucial research topic that we address in this article. Following Collberg et al. (1997), we use the term *potency* to describe how much an obfuscation transformation increases the effort required to understand, analyse, or reverse-engineer protected code. Prior work has approached potency either indirectly, through code-level complexity metrics (e.g. Source

* Corresponding author.
E-mail addresses: leonardo.regano@unica.it (L. Regano), daniele.canavese@cnr.it (D. Canavese), cataldo.basile@polito.it (C. Basile), marco.torchiano@polito.it (M. Torchiano).

https://doi.org/10.1016/j.cose.2026.104881
Received 26 July 2025; Received in revised form 11 December 2025; Accepted 24 February 2026
Available online 6 March 2026

Lines Of Code (SLOC), Cyclomatic Complexity (CC), size and complexity of the Control Flow Graph (CFG), and Halstead complexity measures) computed before and after obfuscation (Tonella et al., 2014), or directly, through empirical studies that measure the success rate and time needed by human attackers to complete reverse-engineering and code-comprehension tasks (Ceccato et al., 2017; Viticchié et al., 2020).

This article falls in the second category. We present the results of a controlled empirical experiment assessing the effectiveness of obfuscation techniques in hindering code comprehension by attackers aiming to compromise assets in software applications. The experiment involved 152 subjects, Master's students in Computer Science Engineering from Politecnico di Torino (see Section 3.6). The subjects performed tasks designed to assess the difficulty of comprehending code obfuscated with two variants of Control Flow Flattening (CFF) implemented by Tigress,[1] a FOSS obfuscator for the C programming language developed at the University of Arizona. One variant employs Opaque Predicates (OPs) to obfuscate the dispatch variable. CFF (Wang et al., 2000) rearranges the basic blocks of the code, so that an attacker cannot readily obtain useful insights on the application's business logic by analysing its CFG. OPs (Collberg et al., 1997) are tautological Boolean expressions used to introduce bogus code that is never executed at runtime, thus increasing the amount of code that an attacker must analyse to understand the application's business logic. A complete description of CFF and OPs is available as supplementary material.

The research questions of the experiment aimed to evaluate the impact of two obfuscation transformations (see Section 3.4) on success rates and time. Furthermore, the experiments sought to highlight the impact of layered protection, i.e., when both techniques are used together to protect the applications. To the best of our knowledge, no prior empirical study has evaluated the impact of combining obfuscation techniques on the success rate or the time required for code comprehension tasks in a quantitative manner. Trying to bridge the gap between objective and subjective approaches to estimate the potency, the experiment attempted to answer two additional research questions, i.e., whether complexity metrics can be used to accurately predict the impact of the treatments on success rates and attack time.

This study has two major novelties.

1. To the best of our knowledge, it is the first study to assess the impact of layering multiple obfuscation techniques on the same protected code.
2. Furthermore, we provide experimental proof of the correlation between objective metrics of attacked code and the likelihood of a successful attack.

For reproducibility purposes, we provide all questionnaires administered to the experiment subjects, their answers, the C source code that constitutes the treatments, the scripts used to analyse the gathered data, and the results obtained. All this information is contained in a replication package available on GitHub.[2]

This paper is organised as follows. Section 2 introduces the background and related work. 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. Section 5 discusses the results we obtained. Finally, Section 6 concludes and outlines future work.

2. Background and related works

Software obfuscation is a family of protection techniques that aim to harden code against reverse engineering, by transforming the application's code to hinder human comprehension while preserving the original application's business logic. Perfect obfuscation has been proven theoretically impossible by Barak et al. (2001). Furthermore, some works have shown that obfuscated code can be recognised (Dalla Preda et al., 2006); hence, these techniques are not stealthy. To the best of the authors' knowledge, no studies in the literature, either theoretically or empirically, demonstrate the effectiveness of these techniques against the reverse engineering of C programs; however, some works exist for Java (Ceccato et al., 2009). Even worse, there is no consensus even on which aspects to consider for evaluating software protection (De Sutter et al., 2024). Nonetheless, they have been extensively researched and employed for a long time in the software security industry to protect commercial software, albeit reportedly uncommon in practice (Becker et al., 2024). It is implicitly assumed that they effectively delay reverse engineering tasks (Chow et al., 2001).

Recent work has applied Artificial Intelligence (AI) to automate reverse engineering tasks (Ghimire et al., 2025) and is beginning to change both sides of the obfuscation/reverse engineering arms race. Large Language Models (LLMs) have been shown to generate effective low-level obfuscations (e.g. dead-code insertion, register substitution) (Mohseni et al., 2025) and to assist analysts in deobfuscating malware samples (Patsakis et al., 2024), while supervised and explainable deep models can recognise obfuscation patterns in binaries (Greco et al., 2024; Conti et al., 2022). Beyond these first applications, several studies have started to systematically assess the code-understanding capabilities of LLMs. Fang et al. (2024) evaluate various LLMs on code analysis tasks showing promising results on obfuscated code explanation. Ma et al. (2024) analyse how code pre-trained models and LLMs capture program syntax and semantics, reporting that semantic understanding remains fragile and highly sensitive to superficial changes in identifiers and structure. AI techniques are also being explored to reverse engineer other opaque artefacts and to scale traditional program-analysis pipelines. For instance, Wu et al. (2022) and Liu et al. (2023) demonstrate that Deep Neural Networks (DNNs) embedded in binaries can be decompiled, recovering the DNNs topologies, parameters, and hyperparameters, enabling adversarial example generation and model stealing. In parallel, He et al. (2021) show how symbolic execution can be enhanced using Machine Learning (ML), in particular leveraging Reinforcement Learning (RL) to tackle the path explosion problem. A similar approach by He et al. (2019) leverages RL to guide fuzzing learning from symbolic execution results. Furthermore, binary similarity frameworks are increasingly leveraging Deep Learning (DL) (Haq and Caballero, 2021; Tian et al., 2021; Yang et al., 2023) to improve accuracy and reduce computation times, using various features to represent the code or the graph representations (e.g. CFG) of the analysed binaries. These developments suggest that future reverse engineers will increasingly rely on AI-driven tooling, and that obfuscation strategies must be evaluated not only against human analysts, as in our study, but also against automated attacks that learn to compensate for syntactic diversity.

2.1. Layered protection

Layered protection is a principle in software protection that suggests applying more than one protection technique to the same piece of code to further increase the effort required for an attacker to carry out their understanding tasks (Xu et al., 2020). To be effective, the protections to apply need to reinforce each other. An example of synergistic techniques is adding anti-tampering techniques (e.g. software attestation Viticchié et al., 2016a) to code protected with obfuscation. Obfuscation is expected to make understanding more difficult, and anti-tampering techniques make writing patches more difficult, as they need to be stealthy to evade the used anti-tampering techniques.

As another example, Udupa et al. suggested using OPs to harden the understanding of the CFF dispatch variable behaviour (Udupa et al., 2005). We deemed this a good example of layered protection, and therefore used it to generate one of the experimental treatments, as described in Section 3.4. Luckily, Tigress supports methods that obfuscate the dispatch variable of the CFF with OPs. For a more thorough presentation

[1] https://tigress.wtf/

[2] https://github.com/daniele-canavese/empirical-obfuscations

of the impact of layered protection, we refer to recently published papers (Basile et al., 2023; De Sutter et al., 2024).

2.2. Measures of potency

In the literature, two main approaches are used to assess the effectiveness 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.

Assessment based on code metrics. One of the early examples of the 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 (Collberg et al., 1997). Relying on the potency definition, Anckaert et al. presented a comparison of obfuscation techniques (Anckaert et al., 2007). Linn et al. considered the confusion factor, which estimates the number of binary instructions that a decompiler is unable to parse (Linn and Debray, 2003). Goto et al. proposed a method to quantitatively measure the complexity of obfuscated code based on compiler syntax analysis (Goto et al., 2000). Udupa et al. estimated the complexity increase in obfuscated programs using data that can be extracted with static and dynamic analysis tools (Udupa et al., 2005). Visaggio et al. instead used code entropy as a protection potency metric for obfuscated JavaScript code (Visaggio et al., 2013). 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 (ANNs) (Canavese et al., 2017). Van den Broeck et al. (2021) proposed a set of software metrics to evaluate the potency of layered obfuscation techniques in protecting software against MATE attacks. Zhao et al. (2020) employed multiple code metrics to train a set of DNNs able to identify the obfuscation techniques used to protect a binary, to ease its deobfuscation. Raubitzek et al. (2024) evaluated the impact of layered obfuscation techniques on code metrics for malware classification purposes. 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 (CSCS), have not been assessed using objective approaches.

Assessment based on controlled experiments with students. Evaluations using experiments with human subjects were introduced by Sutherland et al., who presented the first study in this field (Sutherland et al., 2006). The authors correlated the expertise of attackers with the correctness of reverse engineering tasks. Moreover, they demonstrated that source code metrics are inadequate for estimating delays in attack tasks when binary code is involved. Ceccato et al. measured, in two controlled experiments, the correctness and effectiveness in understanding and modifying decompiled obfuscated Java code, compared to decompiled clear code (Ceccato et al., 2009). This work was extended with a larger set of experiments on several obfuscation techniques in two successive works (Ceccato et al., 2014, 2015). Their replication package was then used by Hänsch et al. to conduct a similar experiment and assess a slightly different set of obfuscations (Hänsch et al., 2018).

Viticchié et al. empirically evaluated with students the attack delay introduced by a data obfuscation technique, namely *variable merge*, reporting that attacks are delayed by a factor of six when that technique is used (Viticchié et al., 2016b). Ceccato et al. experimented with Master's students performing attack tasks on an application protected with CSCS, which moves relevant slices of code to a trusted server, highlighting an effective reduction in attack success rate and an increase in attack time (Viticchié et al., 2020).

Assessment based on experiments with professional participants. Involving professional hackers creates an experimental context that closely resembles a real attack scenario. Unfortunately, they are rare and costly. Also, professionals often do not want to participate in controlled environments or be surveilled at work, leading to a lack of precise measurements and limiting the ability to observe the phenomenon in detail.

Ceccato et al. conducted one of the first empirical evaluations of protection techniques involving professional hackers (Ceccato et al., 2017), asking them to attack fully protected applications (i.e. multiple protections applied to mimic real-world scenarios) to assess how professional attackers perceive and approach these protections. In a subsequent study, the same authors launched a public challenge with a cash bounty to validate the findings from the initial experiment in a broader context with professional white-hat hackers (Ceccato et al., 2018). The papers present methods for coping with these limitations using ad hoc empirical settings.

Open issues. A recent survey highlighted the lack of strength measurements for software protections (De Sutter et al., 2024), arguing that authors of obfuscation tools might consider such measurements irrelevant, too hard or time-consuming to measure. Indeed, there is no consensus on which metrics to use in the objective field, and complexity metrics from the software engineering field may not be computable on obfuscated software, even with the most advanced disassemblers. In the subjective field, the number of experiments required to obtain a wide coverage of applications and protections is probably too large to be affordable. Moreover, some binary techniques can completely prevent the use of certain analysis tools, rendering complex comparisons of alternative attack strategies impossible. Hence, firms developing software protections resort to pen-testing. Furthermore, a framework has been proposed to evaluate the effectiveness of obfuscation techniques (De Sutter, 2025). Despite being in an early stage and mainly theoretical, it looks promising. The results of our experiments can be used to complement and confirm the predictability features.

3. Design of the experiment

3.1. Goals and research questions

The existing literature reports evidence suggesting that obfuscation delays attackers when they have to perform understanding tasks (Viticchié et al., 2020; Ceccato et al., 2014). The assumption is that obfuscation reduces the likelihood that attackers can understand the code's semantics and mount attacks. This hypothesis is also anecdotally confirmed by its extensive use in companies that provide software protection services (Basile et al., 2023).

The *purpose* of this study is to evaluate how different obfuscation techniques can delay or prevent attackers from correctly understanding the code they want to tamper with. The *quality focus* is the ability of the technique to reduce the attackers' capability to mount a successful attack within a useful time frame, by making the understanding phases more complex due to an increased perception of code complexity. The study evaluates the *perspective* of the attackers. In the experiment described in this paper, the role of the attacker is played by students with a consolidated minimum level of expertise in manipulating application source code. The above purpose can be achieved using an experiment aimed at answering the following research questions:

- **RQ1 (treatment-success)**: Do specific obfuscation techniques have a different impact on the possibility for attackers to succeed in understanding an application's code in a given time frame?
- **RQ2 (treatment-time)**: Do specific obfuscation techniques differently impact the time needed for attackers to succeed in understanding an application's code?
- **RQ3 (complexity-success)**: Does the complexity of an application (measured with objective metrics) allow predicting the possibility for attackers to succeed in understanding an application's code with the application in a given time frame?
- **RQ4 (complexity-time)**: Does the complexity of an application (measured with objective metrics) allow predicting the time needed for attackers to succeed in understanding an application's code?

The answers to the above questions are interesting from both theoretical and practical perspectives. Theoretically, they can help specify a more precise formulation of the potency, which was left unspecified in the seminal work by Collberg et al. (1997). Practically, this study will help software developers decide how to protect their applications and what expectations can be reasonably met by adopting obfuscation.

We know from previous studies that obfuscation impacts, to various extents, attackers' activities (Viticchié et al., 2020; Ceccato et al., 2014, 2018). Hence, the first research question concerns the ability of the considered obfuscation techniques to sufficiently hinder comprehension of the protected code, thereby preventing an attacker from mounting a successful attack. In our experimental setting, it means whether, given a certain amount of time, the attack allows the goals to be reached.

In practical terms, we cannot expect protections to prevent any attacker from acting indefinitely. At best, they constitute a hurdle that slows them down. The defenders' purpose is to slow them down so that the attack is not economically viable. Thus, our second research question addresses the techniques' ability to increase the time required to successfully understand the target code in order to mount a successful attack.

The essential idea behind obfuscation techniques is to make understanding a program more difficult. The third research question aims to investigate whether a cognitive complexity measure can be used to predict the success rate of attacks.

Analogously, the fourth research question focuses on whether code complexity can predict the time required to successfully understand the target code, thereby facilitating a successful attack. A positive answer to either of the last two questions could indicate that it is possible to build models that accurately estimate the effectiveness of obfuscation techniques based on code cognitive complexity.

3.2. Threat model

The threat model for this empirical investigation is MATE, which is the reference threat model in this field and has already been used in another empirical study (Viticchié et al., 2020) and a recent work presenting software protection as a risk analysis process (Basile et al., 2023).

Indeed, in our experiment, the attacker has full access to the source code of a software application that can be executed on a local computer without requiring any server interaction. In the real world, attackers mainly have access to binary code. However, we assume that source code, almost as comprehensible as the original source, can be obtained with sufficient effort. Experienced attackers can reconstruct high-quality representations of the original source code thanks to commercial or open-source tools.[3] Since the goal of this research is to evaluate the level of protection that can be achieved with selected obfuscation techniques, we are not interested in assessing the additional delay due to binary code decompilation or disassembling (although some obfuscation techniques can make these tasks harder).

3.3. Objects

The objects of this experiment are three open-source applications:

- `arithmetic`,[4] taken from the `bsdgames` Debian package: a quiz game asking the user to evaluate some simple mathematical computations;
- `number`,[5] another application in the `bsdgames` package, a program that converts Arabic numerals to English (e.g. '42' becomes 'forty two');
- `tictactoe`, the listing 44 in the book C: The Complete Reference, 4th Ed. (Schildt, 2000), an implementation of the tic-tac-toe game where the user can play against the computer.

Practical and methodological considerations guided the selection of these three applications. First, all programs are open-source and implemented in ANSI C. Thus, compiling the source files does not require any specific compiler or flags. The objects can be built on Linux, macOS, and Windows and were tested with the GCC, MSVC, and Clang compilers. The applications can be launched by running the built executable, which presents an interactive command-line interface.

Second, they differ in functionality, covering arithmetic quizzes, string formatting, and interactive gameplay, thus providing some variation in application behaviour. Third, as reported in Table 1, the applications present increasing values of code complexity metrics and SLOC. Moreover, applications needed to remain small enough to ensure that participants could realistically complete the assigned tasks within the allotted time.

Each program is accompanied by a README file that describes the program and provides instructions on how to build and launch it, without indicating whether the version is protected or not.

3.4. Treatments

The treatments in this experiment are the protections applied to the programs (objects) presented in Section 3.3.

- The first treatment (CFF_s) consists of transforming the code using the CFF obfuscation with the *split basic block* Tigress option. This option further divides the original basic blocks to increase the number of `case` branches.
- The second treatment (CFF_{op}) consists of transforming the code using the CFF technique with the *Flatten Obfuscate Next* Tigress option described in Section 2.1. This option uses OPs to obfuscate the code that updates the CFF dispatch variable.
- The third treatment ($CFF_s + CFF_{op}$) is a layered approach that transforms the code using CFF with both the *split basic block* and *Flatten Obfuscate Next* options.

Moreover, all the treatments were also invoked with the Tigress *encode literals* obfuscation,[6] which removes the variables semantics and avoids simple string searches. The Tigress invocation parameters used to obfuscate the literals and the encoder function are reported in the supplementary material.

All the protected treatments exhibit an increase in the values of complexity metrics compared to their vanilla counterparts. The versions protected with a single option have similar complexity values, whereas, as expected, the applications protected with the double CFF show significantly higher metrics values. Table 1 reports complexity metrics values computed with frama-c.[7] Since the three complexity measures strongly correlate with $R^2 > 99.5\%$, we will use only one of them to represent the application complexity. We select SLOC since it is the most intuitive and easiest to compute.

[3] Examples of excellent reverse engineering and exploitation tools are IDA-Pro https://www.hex-rays.com/products/ida/, radare2 https://rada.re/, and Ghidra https://ghidra-sre.org/

[4] https://manpages.debian.org/stretch/bsdgames/arithmetic.6.en.html

[5] https://manpages.debian.org/bullseye/bsdgames/number.6.en.html

[6] This transformation replaces literal strings with calls to a function generating such strings at runtime. Details are available on the Tigress website at https://tigress.wtf/encodeLiterals.html. We obfuscated the functions to encode the literals with CFF to hinder a reconstruction attempt, and moved them into a separate file (`extra.c`), explicitly telling the students to ignore this file to avoid wasting their time.

[7] https://frama-c.com/

Table 1
Complexity metrics of the vanilla and protected applications (SLOC = Source Lines of Code, CC = Cyclomatic Complexity).

APPLICATION	PROTECTION	SLOC	BRANCHES	CC
arithmetic	vanilla	219	33	40
arithmetic	CFF_s	426	96	103
arithmetic	CFF_{op}	361	73	80
arithmetic	$CFF_s + CFF_{op}$	1024	261	268
number	vanilla	274	52	57
number	CFF_s	373	78	83
number	CFF_{op}	349	70	75
number	$CFF_s + CFF_{op}$	629	148	153
tictactoe	vanilla	162	27	33
tictactoe	CFF_s	259	56	62
tictactoe	CFF_{op}	255	54	60
tictactoe	$CFF_s + CFF_{op}$	586	144	150

3.5. Task

We manually altered the source code of the three original applications to introduce a bug that was easily detectable while executing the programs and straightforward to fix once located in the code. The students' task was to find and fix the bug. For the sake of readability, the code snippets in this section contain non-obfuscated literals. We introduced the following errors:

- ($\mathcal{A}_a$) *arithmetic*: we introduced a bit-wise AND between the variable `result` holding the user's answer and a constant, so that the modified application prints a "Wrong result" message also when the user's answer is correct – the task goal is to remove this extraneous operation;

  Listing 1: $\mathcal{A}_a$ buggy version

```
if(atoi(p)==(result&0xF0CACC1A ))
{
    printf("Right!\n");
    ++nright;
    break;
}
printf("Wrong!\n");
```

  Listing 2: $\mathcal{A}_a$ fixed version

```
if(atoi(p)==(result))
{
    printf("Right!\n");
    ++nright;
    break;
}
printf("Wrong!\n");
```

- ($\mathcal{A}_n$) *number*: we introduced a wrong reference in the table `name1` associating numbers with their textual representation that resulted in the wrong print (e.g.2 is translated as `three`) – the task is to remove the wrong reference;

  Listing 3: $\mathcal{A}_n$ buggy version

```
if (*p != '0') {
    rval = 1;
    printf("%s", name1[*p - '/']);
}
```

  Listing 4: $\mathcal{A}_n$ fixed version

```
if (*p != '0') {
    rval = 1;
    printf("%s", name1[*p - '0']);
}
```

- ($\mathcal{A}_t$) *tictactoe*: we modified the logic of the function that checks after each move if the user or the computer won the game so that, if the user wins a game, the variable `matrix` representing the play-board is modified to make the computer win – the task is to remove the code that modifies the play-board.

  Listing 5: $\mathcal{A}_t$ buggy version

```
/* check rows */
for(i=0; i<3; i++)
if(matrix[i][0]==matrix[i][1] &&
   matrix[i][0]==matrix[i][2] &&
   matrix[i][0]!=' ') return
        matrix[i][0]='O';

/* check columns */
for(i=0; i<3; i++)
if(matrix[0][i]==matrix[1][i] &&
   matrix[0][i]==matrix[2][i] &&
   matrix[0][i]!=' ') return
        matrix[0][i]='O';

/* test diagonals */
if(matrix[0][0]==matrix[1][1] &&
 matrix[1][1]==matrix[2][2] &&
 matrix[0][0]!=' ')
   return matrix[0][0]='O';

if(matrix[0][2]==matrix[1][1] &&
 matrix[1][1]==matrix[2][0] &&
 matrix[0][2]!=' ')
   return matrix[0][2]='O';
```

  Listing 6: $\mathcal{A}_t$ fixed version

```
/* check rows */
for(i=0; i<3; i++)
if(matrix[i][0]==matrix[i][1] &&
   matrix[i][0]==matrix[i][2] &&
   matrix[i][0]!=' ') return
        matrix[i][0];

/* check columns */
for(i=0; i<3; i++)
if(matrix[0][i]==matrix[1][i] &&
   matrix[0][i]==matrix[2][i] &&
   matrix[0][i]!=' ') return
        matrix[0][i];

/* test diagonals */
if(matrix[0][0]==matrix[1][1] &&
 matrix[1][1]==matrix[2][2] &&
 matrix[0][0]!=' ')
   return matrix[0][0];

if(matrix[0][2]==matrix[1][1] &&
 matrix[1][1]==matrix[2][0] &&
 matrix[0][2]!=' ')
   return matrix[0][2];
```

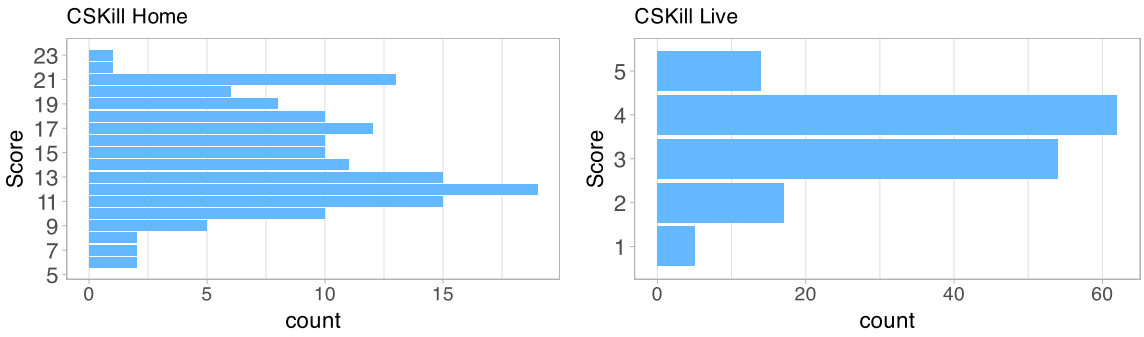
The tasks involved editing the source code to restore the original behaviour. Such modifications are trivial. The most complex part of the tasks was understanding the source code to identify the locations to change. The tasks were designed to assess the difficulty in comprehending the protected code, which represents the area where obfuscation techniques are applied, without requiring significant effort during the tampering phase.

3.6. Subjects

The participants in the study are 152 Master's students in Computer Science Engineering at Politecnico di Torino, selected from those attending the "Computer Systems Security"[8] course, held in the second (and final) year of the degree.

They possess good skills in C programming, thanks to the degree's mandatory courses. Therefore, they can be considered valid candidates to play the role of attackers of the object applications, as in past empirical studies (Ceccato et al., 2018). However, subjects were not expected to have any knowledge about MATE scenarios, attacks, and attack strategies. Indeed, the students involved had not attended any courses on software tampering or software reverse engineering. While some students may have acquired knowledge on such topics through informal means, our primary concern was to ensure that all participants possessed the minimum programming skills necessary to perform the assigned tasks. Since our goal was not to study the impact of expertise on performance,

[8] The 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, firewall and IDS/IPS, Email security https://security.polito.it/~lioy/02krq/



(a) Home skill test results.

(b) Live skill test results.

Fig. 1. Score distribution of the C Skill tests.

we did not attempt to measure or control for advanced background in software security. Moreover, subjects were screened based on their motivation, as participation was voluntary. Participants who performed diligently on all tasks (regardless of success) were awarded 2 extra points out of 30 on the final course exam.

Nonetheless, we evaluated the students' C skills before and during the experiment. In particular, before the experiment, we asked the participants to complete an online C test (*home test*).

The test consisted of 23 multiple-choice questions selected from online resources,[9] covering theoretical aspects of C programming (e.g. pointers, static variables, standard C libraries) and compilers' behaviour (e.g. calls to functions, variables initialisation, stack overflows). Most questions aimed to assess code comprehension abilities (e.g., asking the expected output of a small piece of code).

We opted for a multiple-choice test instead of a programming exercise because we were more interested in evaluating the students' abilities to comprehend and mentally simulate pieces of code, than in assessing their programming proficiency. Indeed, the task was a comprehension one followed by a mostly trivial code change. The overall results of the test are reported in Fig. 1(a). Most students correctly answered at least half of the questions, thus demonstrating that they had sufficient background knowledge to participate in the experiment.

To confirm such results, before starting the experiment, we asked the subjects to answer five additional multiple-choice questions (*live test*), whose results are summarised in Fig. 1(b). The students were able to solve most of the questions in this case.

We also collected self-reported background information *(self-data)* about C programming and reverse-engineering experience. Most (78%) subjects had at least one year of C programming experience, and 20% reported part-time or full-time professional programming experience. All had good experience with one of the supported IDE (Visual Studio, CodeBlocks, Xcode, Eclipse), most (87%) were able to perform basic debugging operations such as setting breakpoints, stepping through execution, and inspecting variables, and more than 80% had at least six months of assembly experience. Reverse-engineering experience was limited: 89% reported less than three months.

3.7. Experimental procedure

The experimental procedure comprises two main phases:

1. the preparation of the experiment, including the preliminary information gathering and the home skill test; in this phase, we have provided subjects with information needed to fill some of the post-experiment reports;
2. the controlled experiment, which consisted of the live C programming test and self-data collection, followed by three code comprehension tasks, one warm-up, and the two main tasks.

In preparing the experiment, we followed the same process as in past research in the empirical assessment of software protection (Viticchié et al., 2016b; Viticchié et al., 2020; Ceccato et al., 2018).

Preparation of the experiment. After presenting the "Hacking Experiment" to the Computer Systems Security course, interested students pre-registered by Email. To keep the setting close to real attack scenarios and to ensure that both the development environment and the participants' skills were adequate, we asked candidates, eleven days before the experiment, to perform a short debugging exercise on their own machine: compile and run a small C program under a debugger and extract a specific integer value. Candidates who submitted the correct value were admitted as subjects and then received the link to the home C programming test.

The subjects were invited to attend a one-hour seminar[10] introducing them to basic information about obfuscation techniques and attack tasks. In particular, we presented an excerpt of a glossary[11] explaining a set of generic attack steps from the ontology published in a previous paper (Ceccato et al., 2018), asking the subjects to read/study it before the experiment. These attack steps were needed to annotate the post-experiment reports, where participants described the procedure they followed when carrying out the tasks.

Controlled experiment. The day before the experiments, the subjects received an Email with three numbered and encrypted zip files containing the three treatments assigned to them. The Email did not include the passwords, which were only made available at the beginning of each task. The purpose was to send a reminder and avoid network overloads on the day of the experiment. The subjects were explicitly asked to follow a well-defined procedure during the experiment:

1. undertake the live C programming test;
2. answer the questionnaire to collect the self-data;
3. perform a warm-up task; and
4. perform the two main tasks.

The *warm-up task*, lasting 40 minutes, asked the subjects to fix a bug in a vanilla application (see Section 3.3). This task aimed to avoid issues (e.g., incompatible environments and problems related to the network and the personal laptops) that could have affected the subsequent two phases of the experiment.

Moreover, we wanted all the subjects to be familiar with the intended task, which, as explained in Section 3.5, consisted of identifying the

[9] We selected questions from https://www.proprofs.com/quiz-school/story.php?title=test-your-c-skills and http://www.pskills.org/c.jsp

[10] The seminar was held nine days before the experiment in the time and room booked for the Computer Systems Security class during the rest of the semester.

[11] https://drive.google.com/file/d/1RIuNkodHIL7QqLGrtNv5NI-Jzm-cZMq7/view?usp=sharing

line of code to modify and applying a straightforward modification that corrected the bug.

At the beginning of this phase, participants received the password to access the first archive and a link to a Google form to submit their answers. Either at the end of the time frame or before, if they were convinced that their task was successful, they were asked to fill out another Google form with the following information: if they succeeded in executing the task; the amount of time in minutes that they needed; the OS, IDE and debugger they used; a qualitative evaluation of the task clarity and difficulty; a description of how they performed the task as a sequence of independent steps, each of them associated with a label (from the glossary), the target variable or function, and a textual description of the activity performed; an archive containing the application source file they fixed and a screenshot showing a successful run of the fixed application.

The *two main tasks*, lasting 70 minutes each, asked the subjects to fix the bug in obfuscated applications. Each participant received one of the two objects they did not receive in vanilla form during the warm-up; one was protected with CFF_s or CFF_{op} and one with $CFF_s + CFF_{op}$, not necessarily in this order. That is, some subjects received the version protected with the double options before a version with a single option enabled. While the participants performed their tasks, the researchers answered all their questions, except those concerning how to carry out the task. We note that, despite careful testing conducted on all platforms, we had to assist the participants in resolving a few setup issues. However, these were mostly limited to the warm-up phase. We further note that participants had no access to AI tools or services throughout the experimental phases.

After the experiment, the actual status of the task (successful or failed) was confirmed by the organisers using a standard assessment procedure:

1. extract the fixed source file and the screenshot showing a successful run of the application from the submitted archive;
2. automatically check the fix's correctness by compiling the modified source code and launching the obtained binary file against a collection of test cases. A manual effort and a collegial decision were devoted to addressing the false negatives, which were subjects who declared a failure but passed all the tests, and the false positives, which were subjects who declared they succeeded but indeed failed the tests. A task was deemed successful only if the source code compiled without errors and all test cases were passed.

3.8. Variables

The research questions presented in Section 3.1 above guided us in selecting the variables to collect during the experiment and formulating the hypotheses.

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

Succeeded: corresponds to the participant's ability to complete successfully or not; it has been assessed as explained in Section 3.7. Variable $Succeeded(s_i, t_i)$ is a Boolean, true if the task t_i by subject s_i was successful.

Time: represents the elapsed time to perform the task. The time is self-reported[12] Variable $Time(s_i, t_i)$ measures the minutes spent by subject s_i to perform the task t_i, regardless of the participant success in performing the task.

We remark that we collected time for all experimental tasks. Because the failed attempts mostly lasted until the end of the allotted time, the analysis will only consider the time for the successful attempts (i.e. when $Succeeded$ == `true`).

The *independent variables* we consider are the following:

Treatment: indicates the type of transformation applied to the source code: { *Vanilla*, CFF_s, CFF_{op}, $CFF_s + CFF_{op}$ }, as described in Section 3.4. The *Vanilla* treatment consists of no obfuscation and has been used in the warm-up task performed during period 1.

Application: indicates the application on which the task was performed. In our experiment, we had three applications: { *arithmetic*, *number* , *tictactoe* }.

Period: indicates the period in which a task was performed. Our experiment consisted of three periods, where period 1 served as a warm-up and was not considered in the hypothesis testing.

Sequence: indicates the specific succession of treatments administered to a given subject, which defines the experimental group the subject was assigned to. The hypothesis testing did not consider the information about the treatment used in period 1.

C Skill Home: indicates the C language skills of the subjects. The score is based on the aggregate results of the (pre-experiment) home test that the participants filled in from home. It ranges from 1 to 23.

C Skill Live: indicates the C language skills of the subjects. The score is based on the aggregate results of the live tests. It ranges between 0 and 5.

$SLOC_{App}$: indicates the size in LOC of the application that was targeted by the task – i.e. after obfuscation has been applied. As discussed in Section 3.4, this metric represents the complexity of the application.

3.9. Hypotheses

Based on the four research questions, we formulate the following null hypotheses to be tested based on the variables defined above:

- H_{ss0}: there is no difference in code comprehension success rate between the two basic obfuscation transformations.
- H_{sl0}: the use of layered protections has no effect in terms of code comprehension success rates with respect to single obfuscation transformation.
- H_{ts0}: there is no difference in code comprehension time between the two basic obfuscation transformations.
- H_{tl0}: the use of layered protections has no effect on code comprehension time with respect to single obfuscation transformation.
- H_{ms0}: there is no difference in prediction accuracy, as far as the code comprehension success rate is concerned, if objective metrics are used together with the combination of application and obfuscation technique.
- H_{mt0}: there is no difference in prediction accuracy, as far as the code comprehension time is concerned, if objective metrics are used together with the combination of application and obfuscation technique.

3.10. Analysis method

The experimental measures are first summarised with basic descriptive statistics. The success rate is reported in terms of both absolute numbers of successful attempts and proportion. For time, we report the mean and standard deviation.

We plan to test the first four hypotheses using frequentist hypothesis testing. Since we adopted a factorial crossover design, as recommended

[12] We cross-checked the self-reported times against the submission timestamp of the Google forms used by the participants to submit their answers, without finding any inconsistencies.

by Vegas et al. (2015), we opted to analyse the data using repeated measures mixed models – both logistic and linear – that included the sequence and period design variables. This choice enabled us to address the potential threats to validity arising from the selected design. We analysed the variance of such a model to check the statistical significance of the factors. We decided not to use non-parametric tests because the sample size – 152 participants for a total of 304 data points – may be sufficient to satisfy the conditions of the central limit theorem (De Winter, 2013); this condition allows us to interpret the results even in the presence of slight departures from normality. Moreover, mixed models have fewer constraints compared to other methods in the case of repeated measures ANOVA.

To test hypotheses about **Success** – i.e. H_{ss0} and H_{sl0} – we use a within-subjects logistic mixed model regression, where the success rate is the dependent variable and the independent variables are the indicator variables for the levels of the variables Treatment, Application, Sequence, and Period. The reference levels (i.e. the intercept) correspond to Treatment = CFF_s, Application = *arithmetic*, Period = 2, and Sequence = $CFF_s \rightarrow CFF_s + CFF_{op}$. The participant ID represents the random component of the model. This model allows us to accurately assess the effect of the main factor in a within-subject crossover design. The first hypothesis (H_{ss0}) will be tested by looking at the statistical significance of the coefficient $\beta_{CFF_{op}}$ corresponding to the effect of adopting CFF_{op} instead of CFF_s. The second hypothesis (H_{sl0}) will be tested by looking at the statistical significance of the coefficient $\beta_{CFF_s+CFF_{op}}$ corresponding to the effect of adopting $CFF_s + CFF_{op}$. In addition to the statistical significance, we will look into the effect size of the Treatment. The $\beta_{\cdot}$ coefficients represent the log-odds-ratio of the application of the treatment level, therefore $e^{\beta_{\cdot}}$ is the odds-ratio.

To test the hypotheses concerning **Time** – i.e. H_{ts0} and H_{tl0} –, we use a within-subjects logistic mixed model regression, where the dependent variable is Time, while the predictors are the same as the previous model. We only considered the participants who successfully completed the task. The hypothesis (H_{ts0}) – concerning the difference between simple obfuscations – will be tested by looking at the statistical significance of the coefficient $\beta_{CFF_{op}}$ corresponding to the effect of adopting CFF_{op} instead of CFF_s. The hypothesis (H_{tl0}) – related to the effect of layering two protections – will be tested by looking at the statistical significance of the coefficient $\beta_{CFF_s+CFF_{op}}$ corresponding to the effect of adopting $CFF_s + CFF_{op}$. In addition to the statistical significance, we will look into the effect size of the treatment. The $\beta_{\cdot}$ coefficients represent the average variation in Time due to the application of the treatment level.

As far as hypotheses concerning the **prediction accuracy** – i.e. H_{ms0} and H_{mt0} –, we build models analogous to the two described above by replacing the indicator variables for Treatment and Application with the size of the application (function). To test H_{ms0}, we observe that the alternative models can be plotted using a Receiver Operating Characteristic (ROC) curve and the Area Under the Curve (AUC) computed. The decision on the hypothesis is based on the comparison of the AUCs using DeLong's test for two correlated ROC curves (DeLong et al., 1988). To test H_{mt0}, we compare the goodness of fit of the two alternative models using both AIC (Akaike information criterion) and BIC (Bayesian information criterion). Since the models are not nested, a Likelihood Ratio test is not possible. The statistical test results are assessed assuming statistical significance at a 95% confidence level (significance level $\alpha = 0.05$). Hence, we reject the null-hypotheses when $p-\text{value} < \alpha$.

All the data processing is performed with the R statistical package.[13] In particular, the mixed model regression was conducted using the `lme4` and `lmerTest` packages (Bates et al., 2015; Kuznetsova et al., 2017), the AUC test using the `pROC` package (Robin et al., 2011).

[13] https://www.R-project.org/

3.11. Threats to validity

We verified the design of our experiment against the checklist of the threats to validity reported by Wohlin et al. (2012), which concern construct, internal, external, and conclusion validity.

Construct validity. First, we report the threats to the *construct validity*, which deal with the relationship between the theoretical constructs and the actual metrics collected for the experiment.

We evaluated the success of a task as a Boolean outcome. While this is a straightforward metric, it reflects a real-case scenario where either the attacker reaches its goals or not within the time frame when assets have a value.

The C programming expertise was assessed using two distinct tests. Both were complex and comprehensive enough to accurately assess students' abilities. Moreover, using C programming tests is standard practice for recruiting programmers used by several companies worldwide (Stepanova et al., 2021). We assured the participants that the test results would only be used for experiment-related purposes. The first test was conducted in an uncontrolled environment, whereas the second was performed in a controlled setting to cross-check the former test's results. In this way, we identified minor anomalies (e.g., students who cheated in the first test) and excluded them. We also noticed that their inclusion would not have changed the results of our analysis.

We allowed the participants to use any tool they were familiar with to complete their task. On the one hand, this approach could represent a confounding factor influencing the results, as a better tool could help complete the task faster and more precisely. On the other hand, we deemed this approach less impactful than forcing the user to use a specific tool with which they could not be very familiar. We tried to mitigate this threat by asking participants to report their strategy, which could reveal the role of tools; moreover, we evaluated the impact using process analysis (see Section A).

Internal validity. Then, we address threats to the *internal validity*, which may affect the ability to capture a cause-and-effect relationship between the independent variables and the experimental outcomes. We must consider all the noise factors that may indirectly influence the outcomes and attempt to mitigate or measure such effects.

Literature reports that professional hackers use quite sophisticated tools to perform both reverse engineering and comprehension tasks (Ceccato et al., 2017, 2018), in particular, in the presence of protected software. Using such tools (e.g. decompilers), they can obtain accurate representations of the code to attack and to reconstruct the source code (almost) completely. Since the participants in our experiment were provided with the source code, we can substantially rule out the need for these tools. Giving the source code is the worst-case scenario – the best case from the attacker's point of view – simulating that accurate reverse engineering tools can recover exactly the original source code.

Another possible threat is that participants may not be aiming for the correct objectives. Right before the experimental activities, researchers explained the tasks and objectives to all participants. Post-experiment questionnaires reported no comprehension issues. Nonetheless, reports indicated that a few participants did not fully digest the provided material. We evaluated these minor misunderstandings and confirmed they did not affect the ability to perform the tasks.

The experiment has been conducted in a single session, thus we can exclude all the common threats related to time and repetitions (e.g. history, testing, mortality, and statistical regression among experiments). A maturation effect may have occurred during the experimental session because each subject was assigned three tasks in sequence. Isolating the first warm-up task was also intended to reduce the maturation effect. Moreover, we have evaluated the impact of maturation between the two main tasks.

Participants were randomly assigned to treatment groups. Since they are all students in the same year and course, we assumed their backgrounds were homogeneous. A posteriori, we verified this assumption;

even if their C skills were not very similar, the distribution of students to treatments resulting from random assignment was balanced.

The number of subjects assigned to each task is not equal. We assumed that the third treatment (layered protection) was the most complex. Thus, we have assigned more subjects to it, to avoid an insufficient number of successful tasks, which would have reduced the relevance and the impact of our analysis. We randomly sampled the data to have a uniform distribution, and the analysis confirms that the unbalanced distribution does not affect the results.

We encouraged the students' participation by granting a 2/30 bonus[14] for the Computer System Security exam grade. We encouraged the participants to do their best by promising in advance that, if they completed all the assigned tasks diligently, they would receive the bonus, regardless of the task's success or the usefulness of the information provided in the report. In our opinion, and based on our experience with previous optional activities conducted with students (Viticchié et al., 2020), offering no incentives at all was not feasible, as students typically do not enjoy spending time on non-profitable academic tasks. Assigning the bonus to all participants led to high participation, but introduced the risk of noise into the collected data, as some subjects might have been primarily motivated by the bonus. Therefore, we added several checks, both to identify subjects who did not score well on the C tests due to their expertise, and to assess the quality of reports, to identify subjects who did not properly document their activities.

External validity. The risks to *external validity* could limit the applicability of the findings, particularly in cases where professional attackers attempt to undermine obfuscated real-world applications.

Professional hackers might be better candidates for assessing the exploitation of MATE attacks; however, involving them presents significant challenges (Basile et al., 2023). Although our participants were students rather than professional attackers, we selected motivated volunteers with a consolidated minimum level of C expertise and designed the tasks and time limits to obtain a sufficient number of successful attacks. Prior work indicates that, in controlled laboratory experiments, top students can be suitable proxies for professionals when the goal is to study relative effects under strictly controlled conditions (Feldt et al., 2018), and several empirical studies on software protection have similarly relied on students as subjects (Viticchié et al., 2016b; Viticchié et al., 2020). Our objects are relatively small compared to typical commercial software, so additional experiments are needed to assess how the studied techniques scale to larger and more complex systems. However, attackers usually invest considerable effort in locating the specific code that must be understood to mount an attack (Basile et al., 2023), and once this code has been identified, its size is often comparable to our experiment's objects (Ceccato et al., 2018). Our results should therefore be interpreted as concerning the comprehension of such localised, attack-relevant code rather than entire applications.

Our study focuses on a specific obfuscation technique, CFF, hardened in one of the treatments by combining it with another obfuscation technique, OPs. Thus, the results presented in this paper may not be generalisable to other obfuscation techniques, both in isolation or layered. Nevertheless, CFF is routinely employed to safeguard software assets, being supported by many code obfuscation frameworks (e.g. Tigress,[15] LLVM Obfuscator,[16] JScrambler,[17] DashO,[18] Allatori.[19]) Furthermore, a recent survey (De Sutter et al., 2024) reports that CFF and OPs are among the most researched code obfuscation techniques in the state of the art.

All participants are enrolled in the identical Master's program. This may lead to a bias in how subjects perform the tasks, linked to the specific educational curriculum implemented at Politecnico di Torino. Indeed, Master's students from different universities may have differing backgrounds in terms of programming languages, frameworks, styles, and methodologies, leading to varying performances when executing tasks.

The study utilised three distinct programs. It is uncertain whether applications with varying structures or from different fields would produce comparable outcomes, even though the architectures identified in our applications are quite typical and present in numerous systems. Our evaluation clearly indicated that the targeted application has an impact on code comprehension time. Additional experiments are necessary to establish a connection between program structures, semantics, and the complexity of the tasks.

As mentioned regarding internal validity, the subjects utilised the application's source code to perform the tasks. This situation does not reflect the scenario of attackers who only have access to the binaries. We recognise that understanding binary code is more intricate than reverse engineering the source code; hence, we can view our situation as a worst-case scenario from the defender's perspective. Attackers may consider various strategies that do not involve an initial attempt to obtain more effective representations of the binaries (such as the source code). In future studies, we aim to assess the differences in attack duration and success rates when using binaries as targets.

Conclusion validity. Finally, *conclusion validity* threats concern the validity of the statistical methods used to derive outcomes from the data. We have used logistic and linear mixed models appropriate for the within-subject crossover design adopted in our experiment and used the suitable tests presented in Section 3.10. We have collected data using survey questionnaires designed according to standard methods and scales (Oppenheim, 1992), and used multiple-choice methods to assess the C skills of the subjects. Tasks were similar and balanced, and subjects were not heterogeneous, as they were all master students; hence, experiments avoided random irrelevance.

4. Results

In this section, we present the analysis results obtained from the data gathered during the experiment, along with the subsequent answers to the Research Questions outlined in Section 3.1.

4.1. RQ1 - Success rate

Fig. 2 summarises the average success rate with an indication of the 95% confidence interval by obfuscation technique (Treatment) and analysed application. In general, we observe that the layered treatment ($CFF_s + CFF_{op}$) lowers the rate of success. The results of the within-subject mixed model logistic regression of success rate vs. obfuscation technique, application, period, and sequence are reported in Table 2.

We draw our conclusions by applying the method reported in Section 3.10. We cannot reject hypothesis H_{ss0}, i.e. no statistically significant difference was found between the two basic obfuscation techniques. On the other hand, we reject the hypothesis H_{sl0}, i.e. the layered application of the two distinct obfuscation techniques has a statistically significant impact on the success rate w.r.t. the basic techniques.

The odds of a successful task when the layered techniques are applied are more than five times lower ($1/e^{-1.69}$) compared to the least effective individual technique (CFF_s) and almost three times lower ($1/e^{-1.04}$) than the most effective one (CFF_{op}). In addition, we observe a statistically significant difference between the Number and TicTacToe applications with respect to Arithmetic, which has twice the odds of a successful task. Concerning the control factors, period, and sequence, we observe no statistically significant effect; this suggests no maturation or fatigue effect was present.

[14] In the Italian University system, grades are assigned on a 30-value scale.
[15] https://tigress.wtf/flatten.html
[16] https://github.com/obfuscator-llvm/obfuscator/wiki/Control-Flow-Flattening
[17] https://jscrambler.com/blog/jscrambler-101-control-flow-flattening
[18] https://www.preemptive.com/products/dasho/features/
[19] https://allatori.com/features/flow-obfuscation.html

Table 2
Mixed model logistic regression analysis for Correctness.

Term	Estimate	Std. Error	z value	Pr(>|z|)	
(Intercept)	1.134	0.601	1.887	0.059	
CFF_{op}	-0.638	0.558	-1.143	0.253	
$CFF_s + CFF_{op}$	1.685	0.431	-3.910	< 0.001	***
$\mathcal{A}_n$	0.870	0.361	2.410	0.016	*
$\mathcal{A}_t$	0.700	0.351	1.996	0.046	*
Period	0.265	0.279	0.951	0.342	
Sequence $(CFF_s + CFF_{op}, CFF_s)$	-0.527	0.522	-1.009	0.313	
Sequence $(CFF_s + CFF_{op}, CFF_{op})$	0.114	0.461	0.247	0.805	
Sequence $(CFF_{op}, CFF_s + CFF_{op})$	-0.261	0.524	-0.498	0.618	

Table 3
Mixed model effects linear regression analysis for Time vs. Treatment and Application.

Term	Estimate	Std.Error	DF	t-value	p-value	
(Intercept)	41.411	4.624	161.674	8.956	<0.001	***
CFF_{op}	3.758	4.074	87.343	0.923	0.359	
$CFF_s + CFF_{op}$	5.953	2.995	97.552	1.988	0.050	*
$\mathcal{A}_n$	0.524	2.692	124.8	0.195	0.846	
$\mathcal{A}_t$	11.148	2.720	124.1	4.099	<0.001	***
Period	-7.515	2.036	87.6	-3.690	<0.001	***
Sequence $(CFF_{op}, CFF_s + CFF_{op})$	0.055	4.524	169.8	0.012	0.9903	
Sequence $(CFF_s + CFF_{op}, CFF_s)$	1.025	3.792	124.7	0.270	0.7874	
Sequence $(CFF_s + CFF_{op}, CFF_{op})$	-2.500	4.545	168.9	-0.550	0.5829	

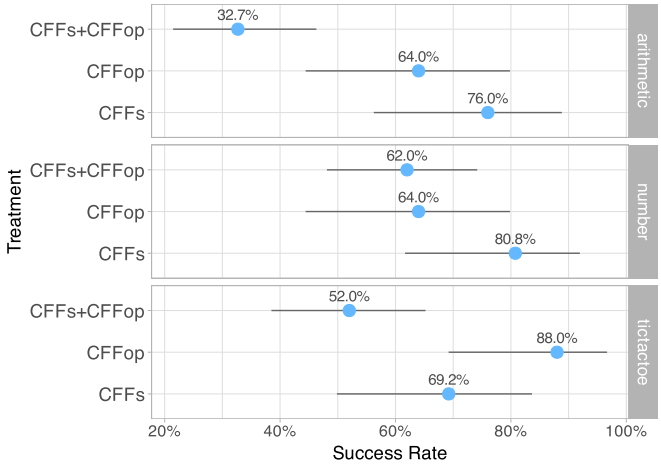


Fig. 2. Success rate by obfuscation technique and application (segments indicate the 95% CI).

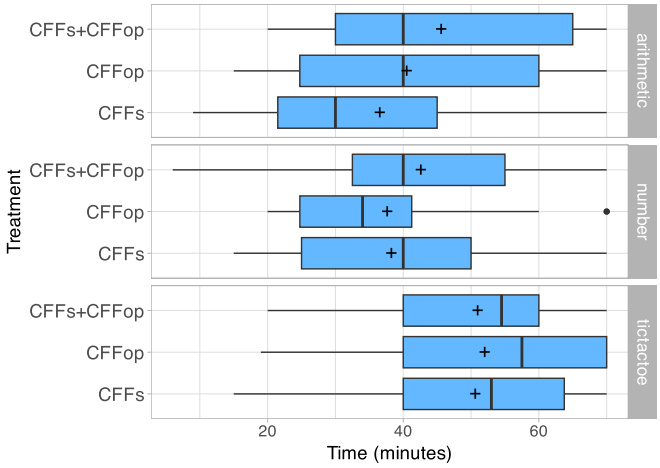


Fig. 3. Code comprehension time by Treatment and Application.

4.2. RQ2 - Time

Fig. 3 reports a series of boxplots that summarise the distribution of time required to complete a successful task by obfuscation technique (Treatment) and by the analysed application. Visual inspection does not allow for observing a clear trend. The mixed model results within-subject linear regression are reported in Table 3.

In this case, we cannot reject the hypothesis H_{ts0} since there is no statistically significant difference between the two single transformations. In contrast, we can reject hypothesis H_{tl0}, i.e., layering the two techniques delays a successful task in a statistically significant way.

On average, the layering of protections increases by six minutes (14% of the time) to complete a successful task. We observed a statistically significant effect of the TicTacToe application, which extended the time required to succeed by 11 minutes on average. Additionally, the period had a statistically significant negative impact, indicating that successful tasks conducted during the last period were 7 minutes shorter on average. This result can be seen as the consequence of a maturation effect. Students became more efficient by completing the previous tasks.

4.3. RQ3 - Complexity and success rate

Fig. 4 illustrates the correlation between code size, expressed in Kilo Lines Of Code (KLOC), and the corresponding average success rate, along with a 95% confidence interval. We observe a clear trend, where larger applications have a lower average code comprehension success rate. The logistic regression results of success rate vs. code size – in place of treatment and application – are reported in Table 4. We observe a statistically significant effect of the code size on code comprehension success. In practice, adding a thousand lines of code reduces the odds of a successful code comprehension task by almost 24 ($e^{3.177}$) times.

4.4. RQ4 - Complexity and time

Fig. 5 shows the correlation between code size, expressed as KLOC, and the distribution of the time required to complete a task. Visually, we cannot see any sensible trend. The results of the linear regression of time vs. code size, in place of treatment and application, are reported in Table 5. The test shows no statistically significant effect of code size on the time needed to complete a task. Only the period has a statistically

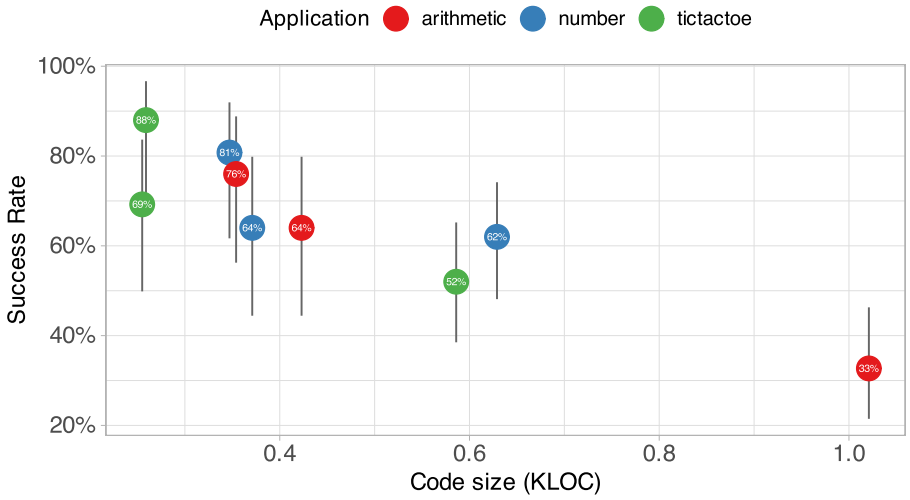


Fig. 4. Success rate vs. Code size.

Table 4
Mixed model effects logistic regression analysis for Success rate vs. Code Size.

Term	Estimate	Std.Error	z-value	p-value	
(Intercept)	1.496	0.828	1.807	0.0707	.
KLOC.app	-3.177	0.653	-4.865	<0.0001	***
Period	0.244	0.277	0.880	0.3788	
Sequence ($CFF_s + CFF_{op}, CFF_s$)	0.228	0.447	0.509	0.6104	
Sequence ($CFF_s + CFF_{op}, CFF_{op}$)	0.390	0.455	0.857	0.3913	
Sequence ($CFF_{op}, CFF_s + CFF_{op}$)	0.255	0.445	0.573	0.5670	

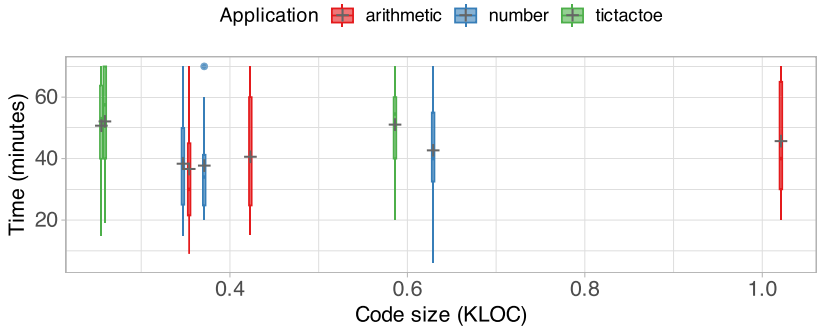


Fig. 5. Code comprehension time vs. Code size.

Table 5
Mixed model effects linear regression analysis for Time vs. Code Size.

Term	Estimate	Std.Error	DF	t-value	p-value	
(Intercept)	47.605	3.746	185.4	12.706	<0.0001	***
KLOC.app	1.530	5.330	117.5	0.287	0.7745	
Period	-8.160	2.331	97.6	-3.501	0.0007	***
Sequence ($CFF_s + CFF_{op}, CFF_s$)	-0.963	3.818	124.9	-0.252	0.8013	
Sequence ($CFF_s + CFF_{op}, CFF_{op}$)	0.358	3.838	109.2	0.093	0.9259	
Sequence ($CFF_{op}, CFF_s + CFF_{op}$)	2.090	3.805	110.8	0.549	0.5840	

significant effect; as far as effect size is concerned, the tasks conducted in the second period appear to be shorter than those in the first by an average of 8 minutes.

4.5. Cofactors: skills

The main co-factor that we considered when balancing the composition of the groups in the experiment was the C language skill. As a confirmatory check, we thus analysed the relationship between the measures of C Skill – C Skill Home and C Skill Live described in Section 3.8 – and the two output variables, success rate and time to complete a code comprehension task.

Fig. 6 reports the success rate vs. the two skill measures, together with a fitted logistic regression curve. We can visually observe a trend that is confirmed by the p-values of the ANOVA: C Skill Home $p = .0319$, C Skill Live $p = .00264$.

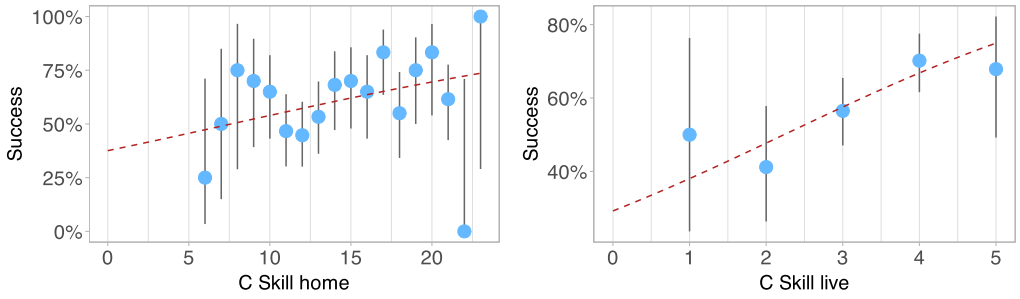


Fig. 6. Success rate vs C Skill.

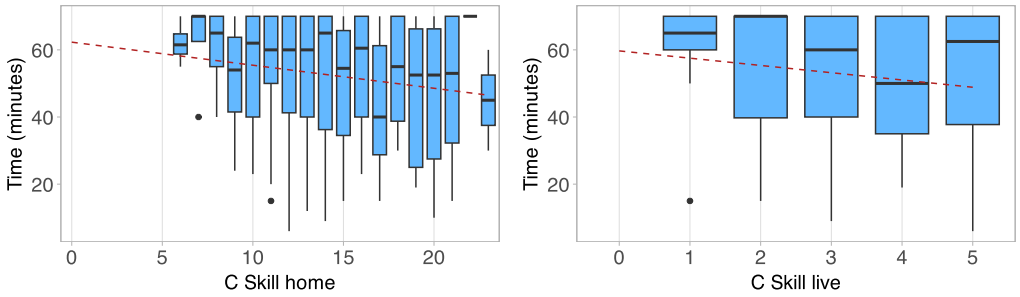


Fig. 7. Time vs C Skill.

Fig. 7 reports the time to complete the task vs. the two skill measures, together with a fitted regression line. We can visually observe a weak trend. The ANOVA's corresponding p-values are $p = 0.0107$ for C Skill Home and $p = 0.056$ for C Skill Live.

In summary, we can confirm that C Skill affects the success rate and partially the time, though in both cases, it cannot explain much; it is just an additional co-factor, which is, in any case, taken into account by the within-subjects models described above.

4.6. Models comparison

The RQs led us to define two distinct models explaining both the code comprehension success rate and time to complete. We compare them here, also reporting two additional models for each output built using the two C Skill measures.

Success rate models. The ROC curves for the alternative logistic regression models – i.e., the ones in Tables 2 and 4 and similar models for the C Skills – for success rate are reported in Fig. 8. The AUC for the two upper curves are 0.897 for the model that considers treatment and application, and 0.894 for the model with code size. DeLong's test for two correlated ROC curves yields a p-value= 0.7224. Therefore, we cannot reject hypothesis H_{ms0}. The performance difference between the two models is quite small and not statistically significant. We also observe that for small values of false positive rate – i.e. 1-specificity – the model based on code size outperforms the one based on treatment and application. The AUC for the C Skill Home model is 0.702, whereas the AUC for the C Skill Live model is 0.711. We can clearly observe from the diagram that these latter two models are statistically different from the former ones.

Time models. The accuracy of the alternative models for time to complete a code comprehension task can be compared using two information criteria, AIC and BIC, reported in Table 6. The lower the value of the indexes, the better the accuracy. The models' accuracy is relatively similar, with a slight edge for the model based on treatment and application.

Table 6
Accuracy information criteria values.

Model	AIC	BIC
Treatment + Application	1576.156	1611.639
Code size	1594.495	1620.301
C Skill Home	2605.213	2646.101
C Skill Live	2607.738	2648.626

4.7. Participant perception

Fig. 9 reports the distribution of participants' answers to three key questions, concerning the clarity of the task they had to perform (TASK_CLEAR), the availability of enough time (ENOUGH_TIME), and the task being easy to perform (TASK_EASY). We observe that the task was clear for a large majority of participants, and they considered the time sufficient in 57% of the tasks. A less clear picture emerges regarding the ease of performing the task, which is consistent with the overall success rate of tasks (61%).

5. Discussion

Our analysis has identified several points worth discussing. Either they shed light on the code comprehension process and obfuscation potency or present interesting aspects that deserve further research, studies, and experiments.

Correlation with the objective metrics. This experiment has provided the first scientific evidence of a correlation between objective metrics, typically used to evaluate code quality, and the comprehension task. By digging deeper into the reasons for the application's influence on both success rates and time, we noted that the success rates and time strongly depend on the code complexity of the applications being tampered with. This correlation was predicted by Collberg et al. (1997), but no evidence was available to support it. Indeed, the analysis has shown that objective metrics are as effective in predicting the success of a task as application and treatment combined. We cannot isolate the metric that has the most important impact, as the treatments on the considered applications increased the values of the metrics, that is, for all the

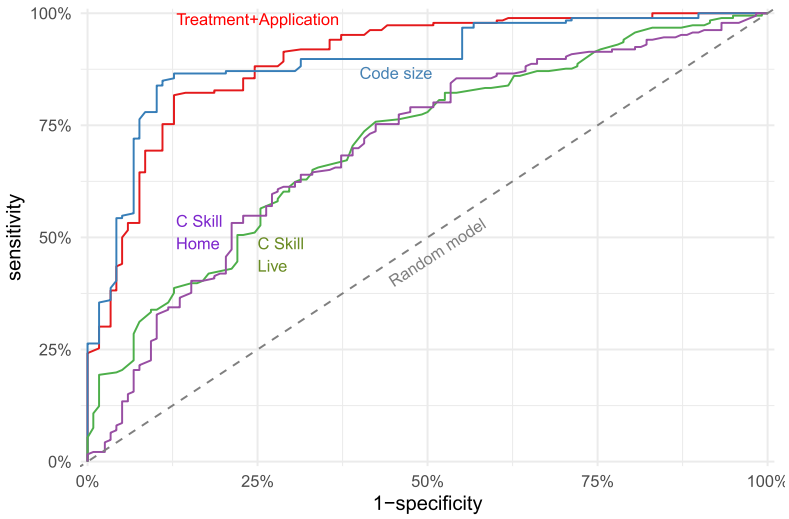


Fig. 8. ROC curves for the Success models.

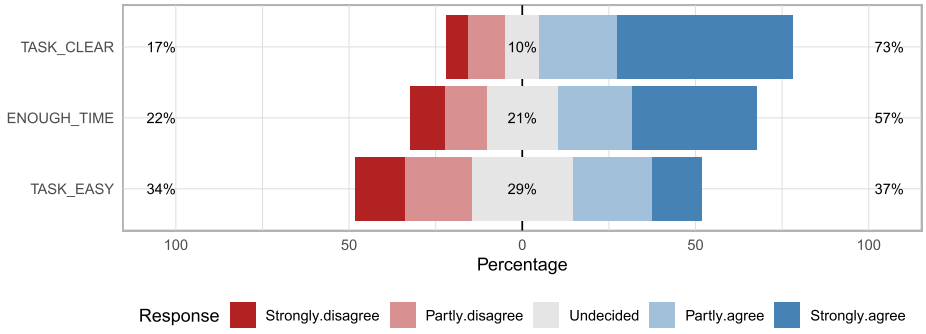


Fig. 9. Participant responses to post-experiment questionnaire.

objects

$$\mu(\text{vanilla}) < \mu(CFF_{op}) < \mu(CFF_s) < \mu(CFF_s + CFF_{op})$$

Dig into the role of the metrics and their relations with the human brain. More experiments would be needed to estimate the individual impacts of metrics. Experiments with more heterogeneous techniques can help determine whether some protections also compel attackers to modify their attack strategies. In particular, it would be interesting to evaluate when attackers abandon static techniques (i.e. reverse engineering tools like Radare2) to use dynamic techniques (e.g. debuggers). Moreover, it would be crucial to determine the characteristics that make code less understandable by correlating them precisely with the physiological aspects of the human brain.

Layered protection. This experiment provides evidence, albeit limited to the specific case of layering two CFF variants on the same C code, to support the current practice of applying multiple protections to the same pieces of code. In our setting, the layered treatment ($CFF_s + CFF_{op}$) reduces the success rate in understanding code in a given time frame by 3–6 times (see Table 2). Moreover, it slightly delays (5%-14%) the subjects who succeed. We cannot determine whether layered protection is effective because of the differences introduced in the protected code by using diverse techniques, or whether it simply increases the complexity metrics more than a single protection. Our intuition indicates that different transformations may impact different aspects of comprehension. Hence, more experiments would be needed to prove this claim, using protection techniques less homogeneous than the ones we used in our experiment (e.g. anti-tampering, data protection, renewability techniques, local vs. remote approaches). Using diverse techniques could also highlight differences in the attack strategies involved by the presence of protections.

App logic may have an impact. The experiments indicated that a specific application, tictactoe, was more complex to tamper with than the increase in complexity could explain. Further experiments are necessary to verify this hypothesis and identify which elements of the application's logic made the task more challenging for the subjects. It would also be insightful to correlate these elements with human cognitive abilities. Moreover, it would be interesting to define experiments that can isolate the task of locating the areas to modify from the complexity of the changes to be made, to consider the task as successful. Indeed, our replication package could serve as a starting point for further investigations into the role of business logic in reverse engineering.

Ensuring enough successes. One crucial aspect to consider when designing controlled experiments is the number of subjects involved and their actual chances of success in the allotted time. The risk is that hypotheses cannot be confirmed or discarded with statistical significance. The fact that only a limited number of subjects succeeded in their tasks limited our ability to precisely correlate time and treatment. Indeed, treatments were chosen among the obfuscation techniques implemented by Tigress, considering the subjects' expertise. The purpose was to allow a reasonable number of successes in the time allotted for the experiment. We manually inspected applications protected by several techniques and discarded those we estimated were too complex.

Subject abilities matter. The results of the C programming tests have a loose correlation with success and time. A trend is visible, but the skill variables are not statistically significant. Involving professional hackers may be necessary if task complexity increases, also considering the need to ensure a proper number of successes.

Subject experience matters. We noticed that in the second period, the subjects are faster (about 7.5 minutes), which can be attributed to a learning/maturation effect. Additionally, some students reported comments like "the code looked similar to the first program.", suggesting that subjects were able to learn something about obfuscation techniques in just a 4-hour session. However, we cannot scientifically determine the extent of this learning based on the available data. Professional pentesters could perform considerably more quickly, which is a concern, as an application is considered compromised as soon as the first attacker cracks it. Correlating the speed of expert pentesters with the results from our empirical experiments involving students may be essential for designing effective protections and assessing the resilience of systems against attacks.

Selecting the right subjects for assessing software protection. Finally, we report that the analysis of the processes, as reported in Section A, indicated that some subjects who failed were probably unable to approach the assigned task, despite the skill tests indicating that all had a sufficient background. This consideration raises a discussion on methods to select more suitable subjects for experiments to assess the effectiveness of software protection and other cybersecurity-related tasks.

6. Conclusions and future work

In this work, we have reported the data collected in a controlled experiment on the effectiveness of layering two widely adopted code obfuscation techniques: CFF and OPs. The experiment involved 152 MSc students as subjects, who performed three different tasks on three different applications. In particular, after executing a preliminary task on a vanilla application, the subjects performed tasks on applications obfuscated with two different versions of CFF and another on an application obfuscated by layering both versions. All tasks required the subjects to understand the obfuscated code, demonstrating their success by fixing a trivial bug in the targeted application.

By analysing the results, we show that that this specific layered protection configuration ($CFF_s + CFF_{op}$) is highly effective in our experimental setting; it significantly correlated with the task's success, reducing the odds of a successful code comprehension by up to 5.4 times with respect to the same application protected with only one of the obfuscation techniques. However, it showed no statistically significant correlation with the code comprehension time. We also noted differences in code comprehension success and time depending on the application being analysed. Moreover, we highlighted a learning effect, as the time to succeed in the second main task was significantly shorter than the time to succeed in the first.

To the best of our knowledge, this is the first empirical experiment to correlate code complexity metrics with the success rate of code comprehension, which constitutes the basis of the software protection potency metric introduced by Collberg et al. (1997). Interestingly, the complexity of an application only filters out the people who succeed, while having a minor impact on the time it takes to complete the task successfully: fewer subjects succeeded, but required approximately the same amount of time.

Following this work, we plan to organise more controlled experiments. First, we assess the impact of layering additional protections, for example, the typical layering of obfuscation over anti-tampering techniques (e.g. remote attestation Viticchié et al., 2016a or code guards Chang and Atallah, 2001), obfuscating the code that verifies and reacts to unauthorised modifications of the protected application. We also plan to involve professional hackers to assess if layering maintains its impact on code comprehension odds when advanced attack tools and techniques are used (e.g. concolic analysis, taint analysis, professional debuggers). Involving expert subjects would help assess the impact of using source or binary code as experimental objects, clarifying the effects of employing students as experimental subjects in empirical assessment studies of software protection.

Furthermore, future studies may leverage larger datasets, comprising programs with increasing complexity and size, to construct regression or ML models that estimate expected code comprehension time or success probability as a function of program size, obfuscation configuration, or code structure. Such models would allow developers to choose protection strategies commensurate with the assets' risk and value.

Future works should also investigate how the increased difficulty induced by code obfuscation relates to subsequent tampering activities performed on the same code, examining whether specific obfuscation strategies (e.g. CFF, OPs) differentially increase the effort required for tampering, and whether this effect persists across attacker profiles, task types, and time constraints. Establishing these relationships would help clarify how obfuscation contributes to tamper-resistance towards cost-effective software protection.

Finally, an additional open question concerns how the emerging wave of AI-driven reverse engineering tools will interact with traditional protections. Recent work show promising results in employing ML, DL and LLMs for code analysis (Fang et al., 2024), deobfuscation (Patsakis et al., 2024), symbolic execution (He et al., 2021), and binary similarity analysis (Tian et al., 2021; Yang et al., 2023). Our results provide a human-centred baseline for the additional attacker effort imposed by code obfuscation; extending the experimental design to include attackers equipped with such AI tooling is a natural next step to assess whether current protections remain effective when confronted with automated, data-driven reverse engineering.

Compliance with ethical standards

Human participants and educational context. The study involved students enrolled at the Politecnico di Torino in Italy. The activity formed an optional component of a course assessment. Participation was voluntary, and students could opt out at any time without penalty.

Informed participation and confidentiality. Students were informed of the aims and procedures in advance. Collected data were restricted to task results and minimal task-related metadata necessary for teaching administration and evaluation. Research analyses used de-identified and aggregate data; no personal identifiers are stored, reported, or published.

Data protection. Data processing complied with the Art. 13 of EU Regulation 2016/679 (General Data Protection Regulation) and the Politecnico di Torino Students' Data Privacy Policy. The University acts as Data Controller; Data Protection Officer contact details are available on the institutional privacy pages, (Fig. A.11).

CRediT authorship contribution statement

Leonardo Regano: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Data curation, Conceptualization; **Daniele Canavese:** Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Data curation, Conceptualization; **Cataldo Basile:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization; **Marco Torchiano:** Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Formal analysis, Data curation, Conceptualization.

Data availability

Data supporting this study are openly available from GitHub at https://github.com/daniele-canavese/empirical-obfuscations.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work was partially supported by project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU. This work was partially supported by ICO, Institut Cybersécurité Occitanie, funded by Région Occitanie, France, and by the European research projects H2020 LeADS (GA 956562), Horizon Europe DUCA (GA 101086308), ARN TrustInClouds, and CNRS IRN EU-CHECK.

Supplementary material

Supplementary material associated with this article can be found in the online version at 10.1016/j.cose.2026.104881.

Appendix A. Process mining

We collected 456 reports describing the attack procedure followed by subjects on their attack tasks. They form the basis used to perform process analysis. This analysis aimed to extract information about the attack strategies and the approaches subjects used when performing their attack tasks. The high-level objective is obtaining confirmations of the results in Section 4 and more insights into how subjects performed their tasks.

More in detail, we wanted to determine if subjects followed general attack strategies when performing attack tasks and if subjects who did not succeed were following a different approach from the subjects who instead succeeded. Moreover, we wanted to understand whether the attack process changed in the presence of the protections (i.e. vanilla applications used during the warm-up vs. protected applications in period 1 and 2) and if these changes were affecting the time required to complete the attack. We also wanted to see if the treatment impacted the process, understand why layered protections were more effective, and the role of complexity metrics.

Finally, we wanted to investigate if the attack process was different depending on the target applications and understand why attacking TicTacToe was more difficult.

A.1. Preparing the reports with a closed coding approach

The reports consisted of a sequence – up to 20 – attack steps, each composed of three fields:

- *assets*, a free text form describing the assets involved in the attack step;
- *description*, an open description of the activities performed, their considerations, what they observed, etc.
- *label*, the categorization of the step based on the fixed list of attack steps in the ontology the subjects received before the experiment.

The sequences of attack steps have been named in the literature as *attack path* (Basile et al., 2019). The reports have been processed using a closed coding approach using the concepts from the ontology provided to the students. This ontology included 32 attack steps from a bigger taxonomy, including 169 concepts, developed to describe hackers while performing attack tasks (Ceccato et al., 2018).

Initially, a collegiate evaluation of the coherence of the data available in the reports was performed. A preliminary analysis allowed evaluating the information in the *assets* field as not useful or of very little usability for our analysis. The reports contained vague indications (just the name of the whole file they modified: useless since only one file was to modify) or erroneous (incorrect names of functions and variables). Therefore, this field was not considered in the rest of the process analysis.

Then, we started processing the *label* assignments and noted incoherencies with the *description* field. A non-negligible number of *labels* were not consistent with the activity described in the free-text description field. We attributed this mainly to a problem of limited clarity of the material provided to the participants, emphasized by the limited subjects' background in software protection; moreover, we cannot exclude that they did not properly study the material provided.

For this reason, we decided to perform an additional step to correct label assignments that were clearly wrong. Every attack step report was independently processed by at least two reviewers, from the authors of this paper[20] Every reviewer could only perform the following changes to the reports:

- Replace the attack step label with another one he was considering more appropriate. For instance, the following comment, "*I fixed the code changing the shift from 47 to 48,*" was initially annotated as "defeat protection". After this review, it was assigned to "tamper with code statically".
- Add an attack step in the attack path when the text implies it. For instance, "*Simply found a strange thing at line 281, I couldn't understand why there was an & 4039822362U [.]. So I removed it, and the program started working.*" was initially annotated as "tamper with data", but an "analyse attack results" step was added, as the subject also assessed the results of the modifications.
- Delete redundant attack steps only when subjects were reporting two consecutive attack steps with the same label to perform the same task on the same asset (instead, the same task on different assets or parts of the applications was considered acceptable). For instance, we found cases of multiple "identify assets by naming scheme" to mark consecutive searches based on file and asset names or "tamper with data" when several variables were changed.
- Mark the entire attack path as incoherent when reports contained severe inconsistencies (e.g. a non-reconcilable sequence of incoherent steps without proper free text), as we refrained from adding any personal judgements or corrections that were not directly expressed in the free text field.

Then, after the independent reviews, the involved reviewers made a bilateral call to find an agreement on the proposed changes and generate the final reports. When both reviewers marked a report as incoherent, it was removed from the rest of the process analysis. In the end, 83 (out of 456) reports were excluded from the next steps of the process analysis: 29/152 of Period 1, 25/152 of Period 2, and 29/152 of Period 3. We observed a similar number of exclusions during the three periods; this means that students did not improve their writing reports during the experiment. They probably needed more experience to be able to fill in the report properly. We included the attack paths in the replication package.

A.2. Process mining on the attack paths

The reports have been analysed with an open-source process mining library (pm4py[21]) that implements a Heuristic Mining algorithm (Weijters et al., 2006). Heuristics Miner is an algorithm that acts on the Directly-Follows Graph, providing a way to handle noise and find common constructs. This algorithm finds a process model that describes the order of events/activities that happen during the execution of a process and has been applied to the attack paths. In practice, given a set of reports, it outputs a graph describing the process.

The process graphs resulting from process mining are formed by:

[20] Stefano Alberto also reviewed reports, but he left our institution before starting to write this paper.

[21] https://processintelligence.solutions/pm4py

- a set of nodes $a \in \mathcal{A}$, where each a is an attack step, associated with a function $\alpha(a)$ to the number of times that a appeared in the report;
- a set of edges $e = (a_1, a_2) \in \mathcal{E} = \mathcal{A} \times \mathcal{A}$, associated with a function $\epsilon(e)$ to the number of times that the sequence $a1, a_2$ appeared in the report.

We have considered three variables to categorize the reports and generate specific process graphs (see Section 3.8):

- *object*, i.e. the three application to tamper with;
- *treatment*, i.e. the three protections used and the vanilla applications that have been provided to subjects during the warm-up experiment;
- the *succeeded* variable (e.g. succeeded, failed, both).

The reports were divided into 3×3 (objects×succeded) + 4×3 (objects×succeded) + 3 (succeeded) = 24 sets, corresponding to all the combinations of objects, treatments, and success information and a set including all the reports.

The pm4py library implementation of the Heuristic Miner algorithm accepts different parameters to fine-tune the process mining by filtering nodes and edges based on noise thresholds. For this analysis, we have used the following parameters:

- DEPENDENCY_THRESH ($\delta \in [0, 1]$), the threshold for the strength of the directly follows dependency, which is used to filter the less evident relations. The higher the number, the more nodes and edges are filtered.
- MIN_ACT_COUNT (α), the minimum number of occurrences of an activity to be considered, used to prune the attack steps that appeared too few times.
- MIN_DFG_OCCURRENCES (ϵ) the minimum number of occurrences of an edge, served to prune the sequences of two consecutive attack steps that appear too few times.

After experimenting with the values, we have decided to use the following combinations

- $(\delta = 0.5, \alpha = 2, \epsilon = 2)$ and $(\delta = 0.5, \alpha = 3, \epsilon = 3)$ for the graphs used by automated procedures, as they are relatively large; for example, the graph for the arithmetic application contains 24 nodes and 81 edges.
- $(\delta = 0.5, \alpha = 0, \epsilon = 0)$, $(\delta = 0.65, \alpha = 0, \epsilon = 0)$, and $(\delta = 0.8, \alpha = 0, \epsilon = 0)$ for manual inspection, as the generated graphs were simple enough to be analysed by humans (see Figs. A.10).

Overall, our analysis included $5 \times 24 = 120$ process graphs, also available in the replication package.

Analyzing the process graphs obtained with the process mining revealed several interesting facts.

It is impossible to infer the existence of a single attack strategy for performing the attack tasks. However, two main attack approaches emerged: the *backward* one, where the subject started from the output and traced back in the source code to the point where the bug originated and the *forward* one, where the subject started from the `main()` function and followed the execution flow (with a debugger) until the bug was found. We manually annotated the attack paths with forward/backward labels; there is no significant evidence to suggest that one approach is better than the other for successfully solving the tasks.

Moreover, we have qualitatively noted that graphs of successful attacks are a little more complex (a few more nodes and edges, a more structured sequence of steps vs. a flatter graph) than the graphs of failed attacks. Moreover, it was evident that the processes followed by those who succeeded looked very similar. Hence, we have introduced a metric (a distance) to measure the similarity between two graphs and confirm our feeling. The metric introduced to measure the graph similarity is based on the Graph Edit Distance (GED) (Sanfeliu and Fu, 1983). GED is defined as the number of edge/node changes needed to make two graphs isomorphic. The computation of the exact value of the GED is an NP-hard problem. For this reason, we used the optimize_graph_edit_distance available in the *networkx* Python library[22] to compute an approximated value of the distance.

Table A.7
GED transformation costs.

Action	Default Cost	Modified Cost
Node n insertion/deletion	1	$\alpha(n)/\lvert\mathcal{A}\rvert$
Edge e deletion/insertion	1	$\epsilon(n)/\lvert\mathcal{E}\rvert$
Node substitution $n_1 \rightarrow n_2$	0	$\lvert\alpha(n_2)/\lvert\mathcal{A}\rvert - \alpha(n_1)/\lvert\mathcal{A}\rvert\rvert$
Edge substitution $e_1 \rightarrow e_2$	0	$\lvert\epsilon(e_2)/\lvert\mathcal{A}\rvert - \epsilon(e_1)/\lvert\mathcal{A}\rvert\rvert$

Table A.8
Similarity index computed on the graphs depending on the app (tictactoe = ttt, numbers = num, and arithmetic = arith) and the succeeded vs. failed tasks (S vs. F).

	ttt (S)	num (S)	arith (S)	ttt (F)	num (F)	arith (F)
ttt (S)	-	0.96	0.97	1.29	1.37	1.43
num (S)		-	0.87	1.06	1.12	1.35
arith (S)			-	1.21	1.31	1.47
ttt (F)				-	1.02	1.07
num (F)					-	1.1
arith (F)						-

The computation of GED can be customized by acting on the costs of graph transformation operations. The cost used for the distance analysis follows these rules (see Table A.7).

- The cost of adding or removing a node is computed as the ratio between the cardinality of the interested node (the number of times an attack step has been used in the report) and the total number of nodes in the graph (the number of attack steps in all the reports).
- The cost of adding or removing an edge is computed as the ratio between the cardinality of the specific edge and the total number of edges in the graph.
- In case of a substitution, the cost is computed as the absolute value of the difference between the cost of the removed entity and the added one.

The similarity index computed on process graphs is shown in Table A.8, divided according to their success. For instance, the process graph obtained from the reports of subjects who successfully tampered with the arithmetic application has a similarity of 0.87 compared to the process graph generated from the reports of subjects who successfully tampered with the number application. On average, two graphs of successful attack tasks are more similar than two process graphs of failed tasks (see Fig. A.12. On average, subjects who succeeded followed similar processes (average similarity 0.93), regardless of the treatment, while the processes of the subjects who failed were less similar[23] (average similarity 1.07). Interestingly, the process of successful tasks was clearly different from that of subjects who failed (average similarity of 1.29), highlighting the importance of following the correct process to approach assets.

The analysis of the processes also revealed that the treatments did not significantly impact the attack strategy. Since the analysis in Section 4 showed that tasks were less successful and slower, and the process remained unchanged, we can infer that the application of obfuscation techniques slowed down the processes, and those that did not succeed may have succeeded with more time. More precisely, understanding the code to locate the proper area to modify becomes more complex. Hence, this is another potential piece of evidence that the correlation with objective metrics may be the leading factor in delaying attackers. However, further experiments focusing on process analysis

[22] https://networkx.org/

[23] Paraphrasing Tolstoy, "succeeding subjects are all alike; every failing subject failed in its own way."

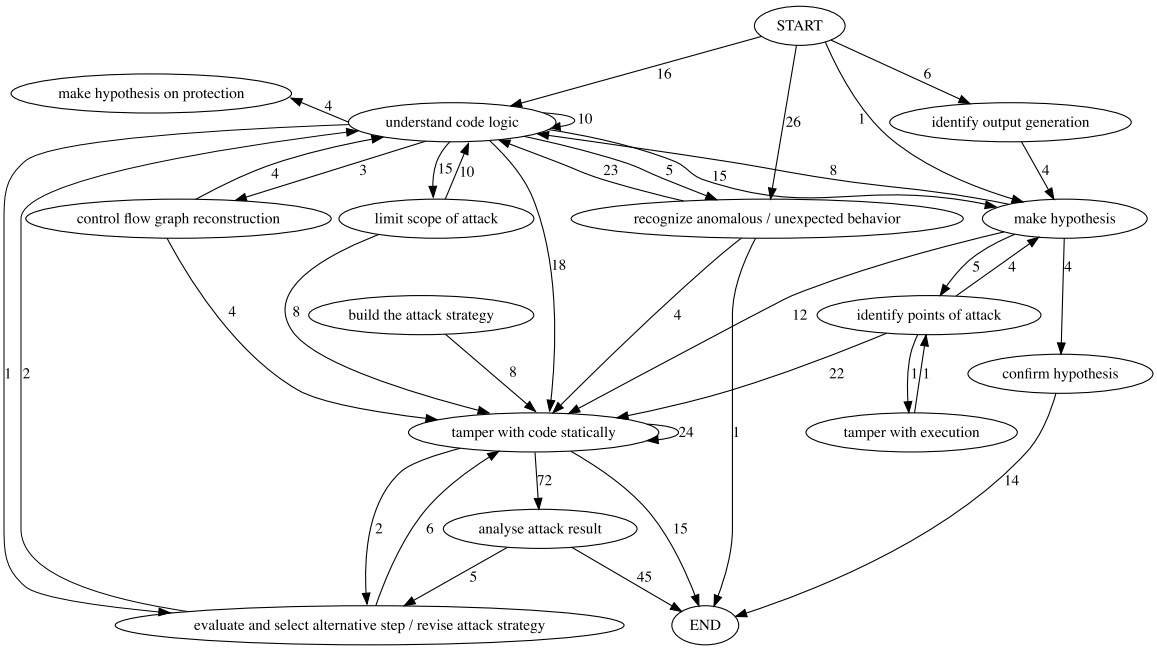


Fig. A.10. Process for succeeded tasks on the arithmetic app ($(\delta = 0.8, \alpha = 0, \epsilon = 0)$).

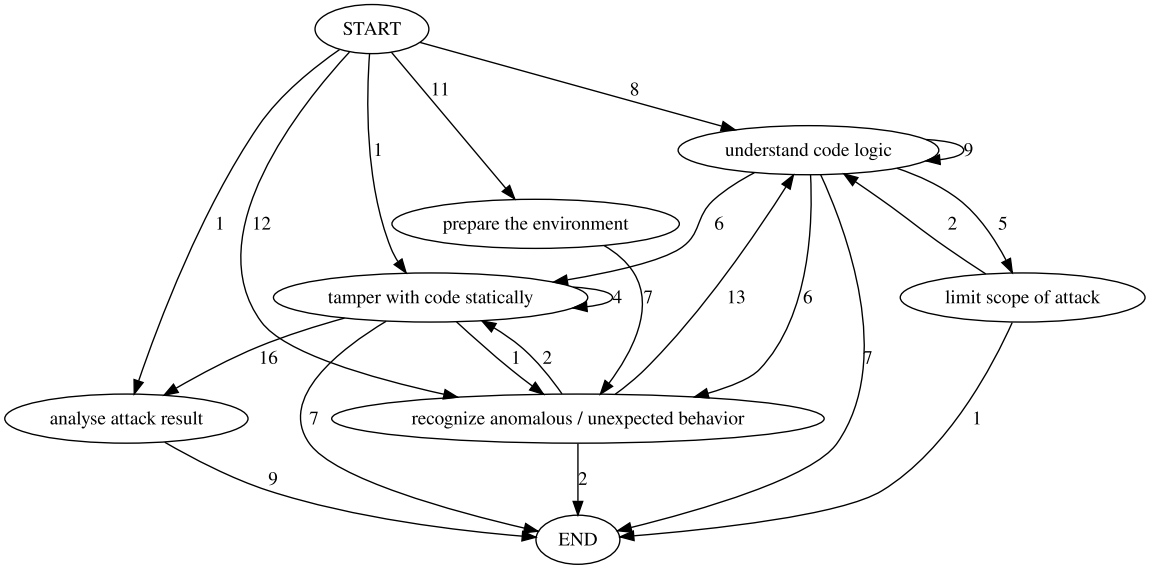


Fig. A.11. Process for failed tasks against the arithmetic app ($\delta = 0.8, \alpha = 0, \epsilon = 0$).

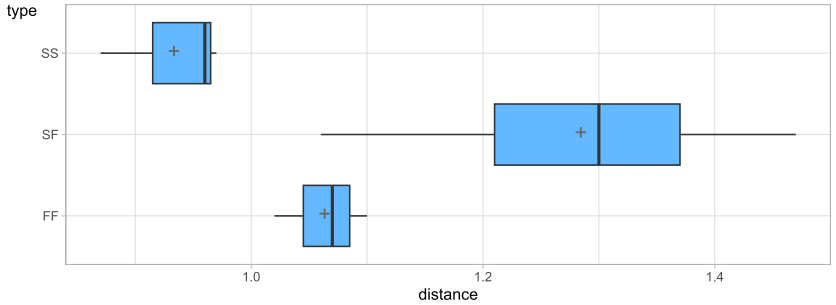


Fig. A.12. Distribution of distances presented in Table A.8.

would be required to make the findings scientifically relevant. Perhaps some students were employing a different strategy, which is why they failed on the protected application.

Moreover, we have found some evidence in the reports that a learning process happened. Indeed, some reports used the step name “recognize similarity with already analysed protected application” and some free text descriptions indicated that they recognized similar patterns from the attack task in a previous period “the code looked similar to the first program”.

Summarizing, we performed an analysis of the attack steps followed by the subjects, which resulted in two additional findings: succeeding subjects typically performed similar attacks, and the used protection did not alter the subjects’ attack strategy.

References

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

Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K., 2001. On the (im) possibility of obfuscating programs. Lect. Notes Comput. Sci. 2139, 19–23.

Basile, C., Canavese, D., Regano, L., Falcarin, P., De Sutter, B., 2019. A meta-model for software protections and reverse engineering attacks. J. Syst. Software 150, 3–21.

Basile, C., De Sutter, B., Canavese, D., Regano, L., Coppens, B., 2023. Design, implementation, and automation of a risk management approach for man-at-the-end software protection. Comput. Security 132, 103321. https://doi.org/10.1016/j.cose.2023.103321

Bates, D., Mächler, M., Bolker, B., Walker, S., 2015. Fitting linear mixed-effects models using lme4. J. Stat. Softw. 67 (1), 1–48. https://doi.org/10.18637/jss.v067.i01

Becker, L., Hollick, M., Classen, J., 2024. SoK: On the effectiveness of control-Flow integrity in practice. In: 18th USENIX WOOT Conference on Offensive Technologies (WOOT 24). USENIX Association, Philadelphia, PA, pp. 189–209.

Van den Broeck, J., Coppens, B., De Sutter, B., 2021. Obfuscated integration of software protections. Int. J. Inf. Secur. 20, 73–101.

Canavese, D., Regano, L., Basile, C., Viticchié, A., 2017. Estimating software obfuscation potency with artificial neural networks. In: International Workshop on Security and Trust Management. Springer, pp. 193–202.

Ceccato, M., Capiluppi, A., Falcarin, P., Boldyreff, C., 2015. A large study on the effect of code obfuscation on the quality of java code. Emp. Soft. Eng. 20 (6), 1486–1524.

Ceccato, M., Di Penta, M., Falcarin, P., Ricca, F., Torchiano, M., Tonella, P., 2014. A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. Emp. Soft. Eng. 19, 1040–1074.

Ceccato, M., Di Penta, M., Nagra, J., Falcarin, P., Ricca, F., Torchiano, M., Tonella, P., 2009. The effectiveness of source code obfuscation: an experimental assessment. In: IEEE 17th International Conference on Program Comprehension (ICPC), pp. 178–187. https://doi.org/10.1109/ICPC.2009.5090041

Ceccato, M., Tonella, P., Basile, C., Coppens, B., De Sutter, B., Falcarin, P., Torchiano, M., 2017. How professional hackers understand protected code while performing attack tasks. In: Proc. 25th IEEE Int. Conf. on Program Comprehension.

Ceccato, M., Tonella, P., Basile, C., Falcarin, P., Torchiano, M., Coppens, B., De Sutter, B., 2018. Understanding the behaviour of hackers while performing attack tasks in a professional setting and in a public challenge. Emp. Soft. Eng. , 1–47.

Chang, H., Atallah, M., 2001. Protecting software code by guards. pp. 160–175. https://doi.org/10.1007/3-540-47870-1_10

Chow, S., Gu, Y., Johnson, H., Zakharov, V.A., 2001. An approach to the obfuscation of control-flow of sequential computer programs. In: Davida, G.I., Frankel, Y. (Eds.), Information Security. Springer, Berlin, Heidelberg, pp. 144–155.

Collberg, C., Thomborson, C., Low, D., 1997. A taxonomy of obfuscating transformations. Technical Report 148. Dept. of Computer Science, The Univ. of Auckland.

Conti, M., Vinod, P., Vitella, A., 2022. Obfuscation detection in android applications using deep learning. J. Inf. Security Appl. 70, 103311. https://doi.org/10.1016/j.jisa.2022.103311

Dalla Preda, M., Madou, M., De Bosschere, K., Giacobazzi, R., 2006. Opaque predicates detection by abstract interpretation. In: Algebraic Methodology and Software Technology. Springer Berlin Heidelberg, pp. 81–95.

De Sutter, B., 2025. A new framework of software obfuscation evaluation criteria. arXiv:2502.14093.

De Sutter, B., Schrittwieser, S., Coppens, B., Kochberger, P., 2024. Evaluation methodologies in software protection research. ACM Comput. Surv. 57 (4). https://doi.org/10.1145/3702314

De Winter, J. C.F., 2013. Using the student’s *t*-test with extremely small sample sizes. Pract. Assess., Res., Eval. 18 (1), 10.

DeLong, E.R., DeLong, D.M., Clarke-Pearson, D.L., 1988. Comparing the areas under two or more correlated receiver operating characteristic curves: a nonparametric approach. Biometrics 44, 837–845.

Falcarin, P., Collberg, C., Atallah, M., Jakubowski, M., 2011. Guest editors’ introduction: software protection. IEEE Software 28 (2), 24–27. https://doi.org/10.1109/MS.2011.34

Fang, C., Miao, N., Srivastav, S., Liu, J., Zhang, R., Fang, R., Asmita, Tsang, R., Nazari, N., Wang, H., Homayoun, H., 2024. Large language models for code analysis: do LLMs really do their job? In: 33rd USENIX Security Symposium (USENIX Security 24). USENIX Association, Philadelphia, PA, pp. 829–846. https://www.usenix.org/conference/usenixsecurity24/presentation/fang.

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., 2018. Four commentaries on the use of students and professionals in empirical software engineering experiments. Emp. Soft. Eng. 23 (6), 3801–3820.

Ghimire, A., Lingala, S.R., Zhang, J., Alsulami, F., Amsaad, F., 2025. A survey on application of AI on reverse engineering for software analysis and security. IEEE Access 13, 152903–152913. https://doi.org/10.1109/ACCESS.2025.3593456

Goto, H., Mambo, M., Matsumura, K., Shizuya, H., 2000. An approach to the objective and quantitative evaluation of tamper-resistant software. In: Third Int. Workshop on Information Security. Springer, pp. 82–96.

Greco, C., Ianni, M., Guzzo, A., Fortino, G., 2024. Enabling obfuscation detection in binary software through explainable AI. IEEE Trans. Emerg. Top. Comput., 1–12. https://doi.org/10.1109/TETC.2024.3439884

Hänsch, N., Schankin, A., Protsenko, M., Freiling, F., Benenson, Z., 2018. Programming experience might not help in comprehending obfuscated source code efficiently. In: 14th Symposium on Usable Privacy and Security, pp. 341–356.

Haq, I.U., Caballero, J., 2021. A survey of binary code similarity. ACM Comput. Surv. 54 (3). https://doi.org/10.1145/3446371

He, J., Balunović, M., Ambroladze, N., Tsankov, P., Vechev, M., 2019. Learning to fuzz from symbolic execution with application to smart contracts. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. Association for Computing Machinery, New York, NY, USA, p. 531–548. https://doi.org/10.1145/3319535.3363230

He, J., Sivanrupan, G., Tsankov, P., Vechev, M., 2021. Learning to explore paths for symbolic execution. Association for Computing Machinery, New York, NY, USA, p. 2526–2540. https://doi.org/10.1145/3460120.3484813

Kuznetsova, A., Brockhoff, P.B., Christensen, R. H.B., 2017. Lmertest package: tests in linear mixed effects models. J. Stat. Softw. 82 (13), 1–26. https://doi.org/10.18637/jss.v082.i13

Linn, C., Debray, S., 2003. Obfuscation of executable code to improve resistance to static disassembly. In: Proc. ACM Conf.Computer and Communications Security, pp. 290–299.

Liu, Z., Yuan, Y., Wang, S., Xie, X., Ma, L., 2023. Decompiling x86 deep neural network executables. In: 32nd USENIX Security Symposium (USENIX Security 23). USENIX Association, Anaheim, CA, pp. 7357–7374. https://www.usenix.org/conference/usenixsecurity23/presentation/liu-zhibo.

Ma, W., Liu, S., Zhao, M., Xie, X., Wang, W., Hu, Q., Zhang, J., Liu, Y., 2024. Unveiling code pre-trained models: investigating syntax and semantics capacities. ACM Trans. Softw. Eng. Methodol. 33 (7). https://doi.org/10.1145/3664606

Mohseni, S., Mohammadi, S., Tilwani, D., Saxena, Y., Ndawula, G.K., Vema, S., Raff, E., Gaur, M., 2025. Can LLMs obfuscate code? a systematic analysis of large language models into assembly code obfuscation. Proceedings of the AAAI Conference on Artificial Intelligence 39 (23), 24893–24901. https://doi.org/10.1609/aaai.v39i23.34672

Oppenheim, A.N., 1992. Questionnaire Design, Interviewing and Attitude Measurement. Pinter, London.

Patsakis, C., Casino, F., Lykousas, N., 2024. Assessing LLMs in malicious code deobfuscation of real-world malware campaigns. Expert Syst. Appl. 256, 124912.

Raubitzek, S., Schrittwieser, S., Wimmer, E., Mallinger, K., 2024. Obfuscation undercover: unraveling the impact of obfuscation layering on structural code patterns. J. Inf. Security Appl. 85, 103850. https://doi.org/10.1016/j.jisa.2024.103850

Robin, X., Turck, N., Hainard, A., Tiberti, N., Lisacek, F., Sanchez, J.-C., Müller, M., 2011. PROC: an open-source package for r and s + to analyze and compare ROC curves. BMC Bioinform. 12, 77.

Sanfeliu, A., Fu, K.-S., 1983. A distance measure between attributed relational graphs for pattern recognition. IEEE Trans. Syst. Man Cybern. SMC-13 (3), 353–362. https://doi.org/10.1109/TSMC.1983.6313167

Schildt, H., 2000. C: The Complete Reference, 4th Ed. McGraw Hill.

Stepanova, A., Weaver, A., Lahey, J., Alexander, G., Hammond, T., 2021. Hiring CS graduates: what we learned from employers. ACM Trans. Comput. Educ. 22 (1). https://doi.org/10.1145/3474623

Sutherland, I., Kalb, G.E., Blyth, A., Mulley, G., 2006. An empirical examination of the reverse engineering process for binary files. Comput. Security 25 (3), 221–228.

Tian, D., Jia, X., Ma, R., Liu, S., Liu, W., Hu, C., 2021. Bindeep: a deep learning approach to binary code similarity detection. Expert Syst. Appl. 168, 114348. https://doi.org/10.1016/j.eswa.2020.114348

Tonella, P., Ceccato, M., De Sutter, B., Coppens, B., 2014. Poster: a measurement framework to quantify software protections. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. Association for Computing Machinery, New York, NY, USA, p. 1505–1507. https://doi.org/10.1145/2660267.2662360

Udupa, S.K., Debray, S.K., Madou, M., 2005. Deobfuscation: reverse engineering obfuscated code. In: Proc. 12th Working Conference on Reverse Engineering (WCRE’05), pp. 10–54. https://doi.org/10.1109/WCRE.2005.13

Vegas, S., Apa, C., Juristo, N., 2015. Crossover designs in software engineering experiments: benefits and perils. IEEE Trans. Soft Eng. 42 (2), 120–135.

Visaggio, C.A., Pagin, G.A., Canfora, G., 2013. An empirical study of metric-based methods to detect obfuscated code. Int. J. Security Appl. 7 (2).

Viticchié, A., Basile, C., Avancini, A., Ceccato, M., Abrath, B., Coppens, B., 2016a. Reactive attestation: automatic detection and reaction to software tampering attacks. In: Proc. 2016 ACM Workshop on Software PROtection. ACM, pp. 73–84.

Viticchié, A., Regano, L., Basile, C., Torchiano, M., Ceccato, M., Tonella, P., 2020. Empirical assessment of the effort needed to attack programs protected with client/server code splitting. Empir. Softw. Eng. 25 (1), 1–48. https://doi.org/10.1007/s10664-019-09738-1

Viticchié, A., Regano, L., Torchiano, M., Basile, C., Ceccato, M., Tonella, P., Tiella, R., 2016b. Assessment of source code obfuscation techniques. In: Source Code

Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on. IEEE, pp. 11–20.
Wang, C., Hill, J., Knight, J., Davidson, J., 2000. Software Tamper Resistance: Obstructing Static Analysis of Programs, Charlottesville, VA, USA. Technical Report.
Weijters, A.J.M.M., van der Aalst, W. M.P., de Medeiros, A. K.A., 2006. Process mining with the heuristicsminer algorithm.
Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2012. Experimentation in Software Engineering. Springer.
Wu, R., Kim, T., Tian, D.J., Bianchi, A., Xu, D., 2022. DnD: A cross-Architecture deep neural network decompiler. In: 31st USENIX Security Symposium (USENIX Security 22). USENIX Association, Boston, MA, pp. 2135–2152. https://www.usenix.org/conference/usenixsecurity22/presentation/wu-ruoyu.
Xu, H., Zhou, Y., Ming, J., Lyu, M., 2020. Layered obfuscation: a taxonomy of software obfuscation techniques for layered security. Cybersecurity 3 (1), 9. https://doi.org/10.1186/s42400-020-00049-3
Yang, S., Dong, C., Xiao, Y., Cheng, Y., Shi, Z., Li, Z., Sun, L., 2023. Asteria-pro: enhancing deep learning-based binary code similarity detection by incorporating domain knowledge. ACM Trans. Softw. Eng. Methodol. 33 (1). https://doi.org/10.1145/3604611
Zhao, Y., Tang, Z., Ye, G., Peng, D., Fang, D., Chen, X., Wang, Z., 2020. Semantics-aware obfuscation scheme prediction for binary. Comput. Security 99, 102072. https://doi.org/10.1016/j.cose.2020.102072

Leonardo Regano received an M.Sc. degree in 2015 and a Ph.D. in Computer Engineering in 2019 from Politecnico di Torino, where he worked as a research assistant for eight years. He is currently an assistant professor at the Department of Electrical and Electronic Engineering, University of Cagliari (Italy). His current research interests focus on software security, artificial intelligence and machine learning applications to cybersecurity, security policies analysis, and software protection techniques assessment.

Daniele Canavese received an M.Sc. degree in 2010 and a Ph.D. in Computer Engineering in 2016 from Politecnico di Torino, where he worked as a research assistant for more than ten years. He is currently a post-doc researcher at the IMATI (Istituto di Matematica Applicata e Tecnologie Informatiche), in Genova (Italy). His current research interests include using artificial intelligence and machine learning techniques for security management, software protection systems, public-key cryptography, and models for network and traffic analysis.

Cataldo Basile is an associate professor at the Politecnico di Torino, from which he received an M.Sc.in 2001 and a Ph.D. in Computer Engineering in 2005. His research concerns software protection, software attestation, policy-based security management, and general models for detecting, resolving, and reconciling security policy conflicts.

Marco Torchiano received the M.Sc. and Ph.D. degrees in computer engineering from Politecnico di Torino. He is a full professor in the Department of Control and Computer Engineering at Politecnico di Torino, Italy. His current research interests include green software, UI testing methods, open-data quality, and software modelling notations. The methodological approach he adopts is that of empirical software engineering.