Programming Experience Might Not Help in Comprehending Obfuscated Source Code Efficiently

Norman Hänsch, *Friedrich-Alexander-Universität Erlangen-Nürnberg;* **Andrea Schankin,** *Karlsruhe Institute of Technology;* **Mykolai Protsenko,** *Fraunhofer Institute for Applied and Integrated Security;* **Felix Freiling and Zinaida Benenson,** *Friedrich-Alexander-Universität Erlangen-Nürnberg*

https://www.usenix.org/conference/soups2018/presentation/hansch

This paper is included in the Proceedings of the Fourteenth Symposium on Usable Privacy and Security.

August 12–14, 2018 • Baltimore, MD, USA

ISBN 978-1-931971-45-4

Open access to the Proceedings of the Fourteenth Symposium on Usable Privacy and Security is sponsored by USENIX.

Programming Experience Might Not Help in Comprehending Obfuscated Source Code Efficiently

Norman Hänsch
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany
norman.haensch@fau.de

Andrea Schankin
Karlsruhe Institute of Technology
Karlsruhe, Germany
schankin@teco.edu

Mykolai Protsenko
Fraunhofer Institute for Applied and Integrated Security
Garching, Germany
mykolai.protsenko@aisec.fraunhofer.de

Felix Freiling
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany
felix.freiling@cs.fau.de

Zinaida Benenson
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany
zinaida.benenson@fau.de

ABSTRACT

Software obfuscation is a technique to protect programs from malicious reverse engineering by explicitly making them harder to understand. We investigate the effect of two specific source code obfuscation methods on the program comprehension efforts of 66 university students playing the role of attackers in a reverse engineering experiment by partially replicating experiments of Ceccatto et al. We confirm that the two obfuscation methods have a measurable negative effect on program comprehension in general but also show that this effect inversely correlates with the programming experience of attackers. So while the comprehension effectiveness of experienced programmers is generally higher than for inexperienced programmers, the comprehension gap between these groups narrows considerably if source code obfuscation is used. In extension of previous work, an investigation of the code analysis *behavior* of attackers reveals that there exist obfuscation techniques that significantly impede comprehension even if tool support exists to revert them, giving first supportive empirical evidence for the classical distinction between potent and resilient obfuscation techniques defined by Collberg et al. more than 20 years ago.

1. INTRODUCTION

In many developed economies, software is a major driver of innovation and industrial growth. To protect their intellectual property, prevent the creation of illegal copies of software and to avoid the unauthorized program flow changes that might benefit the attackers, software vendors employ various software protection techniques. Software protection is also a technique employed by cybercriminals to prevent malware analysis by security researchers.

USENIX Symposium on Usable Privacy and Security (SOUPS) 2018.
August 12–14, 2018, Baltimore, MD, USA.

Software protection can be achieved in multiple ways. Historically, one of the most successful techniques is using specialized hardware, i.e., to disallow access to source or binary by moving it into an external tamper-proof execution compartment [1, 2]. A slightly weaker possibility to achieve software protection is to move only critical parts of software to a trusted processing environment such as a special processor mode [3] or a remote server [4]. While such trusted processing environments are much cheaper than specialized tamper-proof hardware compartments, both techniques incur a significant economical and organizational overhead.

A comparatively cheap alternative to additional specialized hardware is to assume that the attacker will eventually be able to access the code, but that the code is constructed in such a way that it cannot be easily reverse engineered. A central method to deter attackers in this context is *software obfuscation*, i.e., a software transformation that makes the program code harder to comprehend and to analyze. In contrast to many techniques offered in classical software engineering, software obfuscation is a security technique that aims at *inhibiting* software comprehension by attackers. It is the standard means to protect the bytecode of Android apps from analysis today, and it is applied in almost all malware samples spreading in the wild. Understanding the strength of software obfuscation is therefore key both (1) to raise the protection level for software vendors and (2) to help malware analysts to prioritize reverse engineering tasks.

In 2001 Barak et al. [5] showed that *perfect obfuscation* (meaning that a program does not expose more information than can be derived from its input/ouptut behavior) is impossible in general. In practice, most software protection techniques rely on the definition of *obfuscation transformation* provided by Collberg et al. [6], which means that the program's code is made somewhat more obscure by the application of the transformation without introducing a too high performance overhead.

Reverse engineering is always a combination of human ingenuity and tool support. This led Collberg et al. [6] to distinguish between resilience and potency of obfuscating transformations: *resilience* means the ability to withstand an *au-*

tomated deobfuscation attack, while *potency* refers to the grade of "obscurity" for the *human* reverse engineer added by the obfuscation. While formal complexity metrics can help approximate resilience and potency [7], the strength of obfuscation cannot be fully understood without analyzing its effect on program comprehension abilities of real users, a topic which we further study in this paper.

1.1 Related Work

Program comprehension is a mature field in software engineering, where qualitative and quantitative human factors methods have been used to study software comprehension and to evaluate tools [8, 9, 10, 11]. However, there is surprisingly little work on software comprehension in the context of software obfuscation. Ceccato et al. [12] pioneered the area by performing a series of five controlled experiments to measure the influence of code obfuscation on understanding decompiled Java source code. They studied two obfuscation methods, identifier renaming and opaque predicates, and showed that they have a measurable effect on the ability of humans to solve code comprehension and change tasks. In this work, we partially replicate their study and confirm their results. Using a similar experimental setup but with different programs, Viticchié et al. [13] analyzed the influence of the VarMerge obfuscation. Compared with clear code, VarMerge obfuscated code led to significant differences concerning time and efficiency of the attack, but not in correctness.

Although attacker modeling has been identified as one of the fundamental challenges in usable security research [14], user studies in secure programming has focused on the defenders so far. Oliveira et al. [15] showed that security is not a priority in programming tasks and needs additional cognitive effort. Acar et al. [16, 17] analyzed the influence of the documentation that programmers use when writing code. In an experiment with GitHub users, Acar et al. [18] found that correctly fulfilling security requirements is influenced by the years of programming experience, but not by professional status, e.g., student or working programmer. The latter is most relevant to our work as we investigate the influence of programming experience on reverse engineering skills.

1.2 Contributions

In this work, we measure the effect of source code obfuscation on program comprehension skills of human reverse engineers by means of a controlled experiment with 66 participants and advance the insight into the distinction between resilience and potency of obfuscating transformations as defined by Collberg et al. [6]. More specifically, our contributions are as follows:

- Using a slightly different study design, we replicate and validate the results by Ceccato et al. [12], i.e., we provide further experimental evidence that source code obfuscation makes program comprehension significantly harder.
- We provide original insight into the effect of two obfuscating transformations onto the *reverse engineering behavior*. We show that code analysis behavior differs significantly when trying to comprehend the results of an obfuscation method considered to be *potent* in comparison to a method that is considered to be *resilient*. We therefore provide first empirical evidence into the usefulness of these concepts that were defined in 1997 [6].
- To better understand the factors influencing the potency of obfuscation methods, we provide additional original insight into the impact of different programming experience levels on reverse engineering performance and behavior. We show that classical programming experience does not prepare well for the task of comprehending obfuscated code: While experienced participants were much more efficient than beginners when they worked on non-obfuscated code, the gap in efficiency narrowed significantly when given the obfuscated code. Specific obfuscation and debugging experience, however, appears to be helpful.

Overall, if software obfuscation is applied to protect malicious software, our insights may help to improve the education of malware analysis professionals. If obfuscation is used to protect legal software, then our insights may be helpful to evaluate the quality of protection.

1.3 Outlook

After providing background in Section 2, we state the research hypotheses in Section 3 and describe the experimental setup and methods in Section 4. Results are presented in Section 5. We discuss implications and limitations of our study in Sections 6 resp. 7, and conclude in Section 8.

2. BACKGROUND

We first provide background on the obfuscation techniques and code analysis. We further give details on the experimental setup of Ceccato et al. [12] upon which we build.

2.1 Obfuscation

The generally accepted view on obfuscation is based on the notion of program transformation making the code harder to analyze and to comprehend. Obfuscation can be applied at any level of abstraction, be it source code, byte-code or machine code. Here, we focus on source code obfuscation for two reasons: Firstly, source code obfuscation is still common in the context of Java since byte-code can be easily decompiled.[1] Secondly, source code obfuscation has been studied by Ceccato et al. [12], whose work we partially replicate.

One of the most widely used obfuscation techniques is *identifier renaming* where the names of classes, fields and methods, as well as of local variables are changed to meaningless character sequences. Since identifiers are usually carefully selected to reflect their semantic meaning, removal of this information complicates the process of code comprehension. *Name overloading* [19] extends identifier renaming by using the same names for multiple different entities. We use name overloading as the first obfuscation technique in our study and abbreviate it by NO.[2]

Name overloading does not change the structure of the code. The obfuscation technique of *opaque predicates* (abbreviated as OP) can be used to alter the program's execution flow. A predicate is called *opaque* if its outcome is known at obfuscation time but is hard to deduce by the reverse engineer [19].[3]

[1] While the new Android Runtime (ART) supports also the distribution of native code, using classical bytecode is still common because of backwards compatibility.
[2] Ceccato et al. also used NO but called it "identifier renaming" and used the abbreviation IR, see also footnote 6.
[3] Examples of true and false opaque predicates are $(x^2 + x)\ mod\ 2 = 0$ and $x^2 + 1 \leq 0$ respectively for any real value x.

Opaque predicates can be used to extend existing branches or to insert dead code. We use OP as second obfuscation technique in our experiments. Appendix A provides code examples to illustrate both obfuscation techniques.

2.2 Code Analysis and Eclipse

The process of understanding of the program's code and its key features is referred to as *code analysis*. *Static* code analysis does not involve actual execution of the program, whereas in *dynamic* analysis, code is at least partially executed. Usually, code analysis is supported by tools. For the purpose of Java source code analysis, the Eclipse IDE can be utilized. It can perform both static and dynamic analysis. In the following we briefly describe the capabilities of this tool, as it was used by participants in our study.

The Eclipse IDE supports static analysis by providing additional information for the source code, such as showing the class inheritance hierarchy or call graph, or highlighting cross-references. It can also automate standard modifications of the program performed by the analyst, such as renaming of variables, methods, and fields, or moving methods from one class to another. In this paper we refer to the latter operations as *advanced Eclipse commands*.

The Eclipse IDE also supports program execution in the *debugging* mode. Using this mode, the analyst can perform single-stepping, executing only one instruction at a time, set breakpoints, watch the variable values, and so on. This functionality can be useful to follow the execution of the code under analysis, in order to better understand the dependencies between code and external program's behavior, or to identify predicates suspicious of being opaque, namely those that always have the same value at runtime.

2.3 Obfuscation Studies by Ceccato et al.

Ceccato et al. [20, 21, 12] conducted a series of experiments using two programs and letting participants solve two code comprehension tasks and two code modification tasks for each program. The experiments varied in the type of obfuscation, the type of students (bachelor, master or PhD) and the universities in which they took place, while the experimental tasks remained the same. Since the results of all experiments are summarized in one single paper [12], we refer to this paper in the following.

One of the programs (called *Race* in the following) is an online game that lets two players conduct a car race. Another program, called *Chat*, lets people have public or private online conversations. The programs were given to the participants as source code decompiled from Java bytecode, as this is the usual way how the reverse engineers work on Java code. Depending on the experiment, the programs were provided in different variants: as clear code (unobfuscated), obfuscated with identifier renaming (which was in fact name overloading), or obfuscated with opaque predicates. General software metrics for both programs presented in Table 1 show that the programs are comparable in their complexity. Although Race has a higher number of methods and lines of codes (LOC) than Chat, it has a lower overall cyclomatic number [22] (roughly corresponding to the number of linearly independent paths in a function's code).

Ceccato et al. [12] conducted five experiments that cumulatively evaluated whether code obfuscation influences performance of reverse engineers: Do people solve code comprehension tasks slower and less correct on obfuscated code? If yes, which of the obfuscation methods (NO or OP) reduces the performance more severely? The code comprehension tasks from the study can be found in Table 2.[4] Overall, Ceccato et al. found statistically significant differences only for the obfuscation technique NO, supporting the belief that opaque predicates help to slow down automated analysis rather than performance of human reverse engineers.

	Race			Chat		
Metric	*Clear*	*NO*	*OP*	*Clear*	*NO*	*OP*
Classes	14	14	14	13	13	13
Methods	109	109	125	72	72	88
LOC	1215	1215	3783	1030	1030	3642
$\sum$ Cyclomatic	244	244	1131	253	253	1775

Table 1: Software metrics calculated for the different versions of the programs. Due to the nature of NO, the metrics are the same for NO and Clear.

Task	Description
Race: Box	In order to refuel the car has to enter the box. The box area is delimited by a red rectangle. What is the width of the box entrance (in pixel)?
Race: Laps	When the car crosses the start line, the number of laps is increased. Identify the section of code that increases the number of laps the car has completed (report the class name/s and line number/s).
Chat: Messages	Messages going from the client to the server use an integer as header to distinguish the type of the message. What is the value of the header for an outgoing public message sent by the client?
Chat: Users	When a new user joins, the list of the displayed "Online users" is updated. Identify the section of code that updates the list of users when a new user joins (report the class name/s and line number/s).

Table 2: Participants' tasks for the study of Ceccato et al. [12] and ours.

3. HYPOTHESES

We formulate research hypotheses that aim at answering the following research questions:

- Can we validate the results of Ceccato et al. concerning code *comprehension*?
- Does obfuscation influence the code analysis *behavior* of attackers?
- Does programming *experience* influence code comprehension and behavior of the attackers?

[4] Ceccato et al. also investigated code change tasks that we do not consider in our study. We present differences between their and our study in more detail in Section 4.

3.1 Code Comprehension Hypotheses

Considering the effect of code obfuscation on code comprehension, we evaluate the following hypothesis:

> Obfuscated code is more difficult to comprehend than clear code.

However, the term "obfuscated code" can be instantiated in many different ways. To evaluate such a hypothesis it would be necessary to investigate the effects of a "representative" set of obfuscation methods and it is not entirely clear what this could be. We therefore focus on the effects of the two obfuscation techniques NO and OP and conduct partial replication of prior work by Ceccato et al. [12].

Ceccato et al. found no significant difference between correctly comprehending clear code and code obfuscated with any of the two obfuscation techniques. For the efficiency the results differed. While no significant difference in the efficiency between working on clear and OP-obfuscated code was found, efficiency of working on NO-obfuscated code significantly decreased compared to working on clear code. Similarly, only working on NO-obfuscated code took significantly longer than working on clear code. Ceccato et al. [12] therefore rejected several of their hypotheses concerning OP-obfuscated code. However, since the number of participants in the various studies was quite small (10 to 22), we assume that some effects of the obfuscation methods might have been missed. Therefore, for code comprehension we formulate the same set of hypotheses as Ceccato et al. [12], where capitalized words set in italics indicate independent variables for the statistical analysis:

HC1$_{\mathbf{NO}}$ *NO-obfuscated* code is more difficult to comprehend than *Clear* code.

HC1$_{\mathbf{OP}}$ *OP-obfuscated* code is more difficult to comprehend than *Clear* code.

These hypotheses attempt to approximate the hypothesis on the general effect of obfuscation presented above.

Following the discussion on the *potency* of obfuscation methods [6], i.e., the differing grades of "obscurity" for the *human* reverse engineer added by the obfuscation, the next hypothesis aims at insights into the effects of conceptually different obfuscation techniques. Ceccato et al. found that understanding NO-obfuscated code is more difficult than understanding OP-obfuscated code. However, this difference was statistically significant in only one of two experiments that they conducted with this goal. We seek to validate their results with the following hypothesis:

HC2 *NO-obfuscated* code is more difficult to comprehend than *OP-obfuscated* code.

Ceccato et al. also report that participants with higher experience (measured by their study degree: bachelor, master or PhD student) performed slightly better on both, clear and obfuscated code (the results were not statistically significant). We therefore formulate the following hypothesis:

HC3 The higher the experience of attackers, the easier they comprehend *Clear* and *Obfuscated* code.

3.2 Code Analysis Behavior Hypotheses

Ceccato et al. did not investigate behavior of attackers in solving their tasks. However, they asked participants some questions about their analysis behavior in a post-experimental question, e.g., which percentage of the task time they spent reading the code, or how many program executions in debugging mode they used. They report some (mostly not statistically significant) differences in the answers for clear and obfuscated code. We take their investigation as an inspiration for looking at the *actual* attacker behavior.

In practice, the first step in code comprehension of obfuscated code is usually to identify the particular obfuscation technique and perform experiments with tools for automatic deobfuscation [23]. It is therefore to be expected that comprehension of obfuscated code results in different code analysis behavior from classical reverse engineering, namely that behavior attempts to first identify the obfuscation method or performs simple deobfuscation tasks. In general, we therefore evaluate the following hypothesis:

> Code obfuscation significantly changes code analysis behavior in comparison to analysis behavior for clear code.

Since code analysis behavior appears to target the obfuscation method first, we expect to find differences not only between clear code and obfuscated code in general, but also differences in the behavior between code obfuscated by different obfuscation techniques. We therefore explore the novel behavioral research question by evaluating the following hypotheses with regard to the *behavior* of the attackers for code comprehension tasks:

HB1$_{\mathbf{NO}}$ When analyzing *NO-obfuscated* code attackers behave differently than when analyzing *Clear* code.

HB1$_{\mathbf{OP}}$ When analyzing *OP-obfuscated* code attackers behave differently than when analyzing *Clear* code.

HB2 When analyzing *NO-obfuscated* code attackers behave differently than when analyzing *OP-obfuscated* code.

Since comprehending obfuscated code in practice seems to require additional expertise, we also formulate hypotheses concerning the influence of experience, as previously done for code comprehension:

HB3 Experienced attackers behave differently than beginners when analyzing *Clear* and *Obfuscated* code.

3.3 Measurements

We now describe how we measured code comprehension, behavior and experience.

3.3.1 Code Comprehension Measurements

We measure code comprehension in exactly the same way as proposed by Ceccato et al. [12]:

- *Correctness* (measured per program) is the number of correctly solved tasks: 0 if no task is solved correctly, 1 if precisely one task is solved correctly and 2 if both tasks are correct.
- *Time correct* (measured per program in minutes) is the time spent on average for correctly solving tasks for a program. It is computed as the sum of times spent on correctly solved tasks divided by the number of correctly solved tasks. If no answer was given correctly, the participant was taken out of the calculations.
- *Total time* (measured per program in minutes) shows how long a participant worked on the program, independently on the correctness of solutions. Although this is not a

code comprehension variable by itself, we use it to derive the notion of efficiency below.
- *Efficiency* (measured per program) is *Correctness* divided by *Total time*.

For the tasks that ask to point out a line number where a certain action is performed ("Race:Laps" and "Chat:Users" in Table 2), we evaluated the *Correctness* of the participants' answers in a different way than Ceccato et al. [12]. Whereas they accepted only one specific line number as correct answer, we have adopted a less restrictive interpretation that allowed the following solutions:

- The exact line according to Ceccato et al. [12].
- The line number of the corresponding function header, or the lines interval of the whole corresponding function.
- The exact line of the corresponding function's call site.

We think that all three answers provide a sufficient proof of the participant's understanding of the code functionality. In the hypotheses testing in the sequel, we consider Ceccato et al.'s evaluation for comparison. Two other tasks ("Race:Box" and "Chat:Messages") were evaluated exactly as by Ceccato et al.

3.3.2 Behavior Measurements

To record actions performed by participants during code analysis, we use the Eclipse plugin Fluorite [24] that creates an XML-log of all commands and events with the corresponding timestamps. This data allows us to reconstruct the reversing procedure of each participant with high precision. We extract the following information from the logs:

- The number of the *file open* operations, which correspond to either opening a new file or switching the focus to the already opened one.
- The number of executed *advanced commands* such as automatic identifier renaming, construction of call graphs and type hierarchies.[5]
- The number and the total time of program *executions*.
- The number of times and the total time of the program being in *debugging mode*.
- The total time of *code reading*, which is defined as the overall processing duration for the given program minus the execution and debugging time.

For each action, i.e., program execution, debugging, file open and advanced command, the start and the end timestamp relative to the begin of the program processing are used.

3.3.3 Experience Measurements

Cecatto et al. evaluated the experience of the participants based on whether they were bachelor, master or PhD students. They argued that this is a reliable measure since the authors were in charge of the participants' courses at the corresponding universities [12]. For our study we assume that the participants might have studied at different universities before. Moreover, the attended courses can greatly differ at our university due to different study programs. Further, Acar et al. [18] found that even differentiating between students and non-students showed no significant differences in their participants' skills. We therefore evaluated experience using a more general explorative approach.

[5]Advanced commands have the command ids starting with `org.eclipse.jdt.ui.edit.text.java`.

Individual differences in programming skills, programming experience or experience in dealing with obfuscated code may influence the performance of participants and their analysis behavior. *Experience* relates to the hypotheses HC3 and HB3 and is measured as follows:

- *Programming Experience* is measured on a scale from 1 to 4 using the following question in the pre-study questionnaire: *"How would you describe the quality and the type of the code you wrote so far?"* This question originates from Ceccato et al. [12] and has the following answer options:
  1. Few and small programs (e.g., course exercises)
  2. Many small programs
  3. Small programs and 1 or 2 big programs (e.g., thesis and projects)
  4. Big programs
- *Study-relevant Experience* refers to the experience and knowledge in code obfuscation, Java, the usage of Eclipse for software development, debugging software, the usage of Eclipse for debugging software. These factors are measured using questions *"Please indicate your experience with ..."* in the pre-study questionnaire on a 5-point Likert scale with values from 1 = *very low* to 5 = *very high*;
- *Comprehension Skills* are measured by considering the efficiency of a participant when working with *Clear* code.

4. METHOD

In this section we outline study materials and design, including ethical considerations, and describe recruitment and demographics of the participants. Finally, data analysis techniques are presented.

4.1 Study Materials

4.1.1 Code and Questionnaires

Ceccato et al. [12] provided us with original .jar-files for the clear code of the Chat and Race programs used in their studies. We obfuscated the source code of both programs (Chat and Race) either with name overloading (NO) or with opaque predicates (OP) using the SandMark tool [25] which was reportedly also used in previous work.[6] The resulting three .jar files were decompiled using JAD [26], leading to three source code versions of each program: two obfuscated versions (NO and OP) and the unobfuscated original version. These were used by the participants in our study.

We used the questionnaires by Ceccato et al. [12] that were slightly adapted for our study. For example, we did not ask the participants to estimate the number of code executions per task, since we could measure this in our setup. The questions asked in the survey, their order and under which circumstances they were presented to the participants can be found in Appendix B.

4.1.2 Technical Setup

The technical setup of our study was designed to be especially easy and efficient to replicate. We prepared virtual machines equipped with the Eclipse IDE for analyzing the

[6]While Ceccato et al. [12] claim to have investigated the effect of *identifier renaming* (IR) using SandMark, SandMark does not explicitly offer this obfuscation method. So while we were able to reproduce the obfuscated version of OP, we could not reproduce the code for IR. We therefore chose the "closest" obfuscation variant to identifier renaming provided by SandMark which was *name overloading* (NO).

programs and with the Firefox browser for filling out the online questionnaires. All questionnaires and the code comprehension tasks were combined into one online questionnaire that was developed with LimeSurvey[7]. Participants therefore did not have to change the medium they work on. This also ensured that the participants did not forget to answer the questions, as they could not proceed to the next task otherwise. We were also able to take more precise time measurements than the previous work [12], where the participants filled in the questionnaires on paper and wrote down start and end time of each task.

4.2 Participants

The participants were recruited at an engineering department of a German university. The recruiting materials (flyers, posters and emails) required the participants to have at least basic knowledge of Java and Eclipse.

In total 76 participants took part in our study (8 female). For the evaluation, data of 10 participants were excluded from the analysis because they indicated in the survey that they did not have enough time to successfully complete all tasks. This leaves a total of 66 participants. Most of them (44) were bachelor students, 20 master and 2 PhD students. Ages ranged from 18 to 31 with an average of 22 years.

Most participants were studying computer science (40), followed by computational engineering (4) and medical engineering (4). Furthermore, 16 participants (24.2%) stated that they already participated in a course related to software obfuscation, 7 participants stated that they already worked full-time as a programmer. Part-time working experience was reported by 16 participants.

Concerning previous coding experience, 34 participants (51.5%) stated that they already wrote one or two big programs. The two groups who either only worked on few small programs (19.7%) or on many small programs (21.2%) were almost equally represented. Participants with high experience in big programs made up 7.6% of the participants.

4.3 Study Design

4.3.1 Experimental Setup

Our experimental setup is slightly different from Ceccato et al. [12]. The main differences are summarized in Table 3. Whereas in their work, each participant attended two sessions on two different days in order to reduce the fatigue effects, we opted for having only one session per participant, because a simplified study design allowed us to recruit more participants and thus obtain more results for robust statistical analysis.

To reduce the fatigue effects in our study, we reduced the number of tasks on which each participant worked. For each program, the participants worked on the two comprehension tasks from the original study (Table 2). The two additional change tasks given by Ceccato et al. [12] were omitted.

Moreover, Ceccato et al. [12] used the *within subjects design* [27, 28] where each participant worked on all tasks for a particular study. For example,when they compared between clear code and OP, all 16 participants worked on clear and OP-obfuscated code. In the study where the influence of OP and NO were compared, all participants performed tasks on programs obfuscated with NO as well as with OP. This design is especially useful for small numbers of participants.

[7] https://www.limesurvey.org

	Ceccato et al. [12]	this paper
Sessions per participant	2	1
Number of tasks per program	4	2
Participants (Clear vs NO)	10 and 22[1]	31
Participants (Clear vs OP)	16	35
Participants (NO vs OP)	13 and 13[1]	66
Participants (total)	74	66

Table 3: Experimental setups by Ceccato et al. versus this work. Due to different study designs (*within subjects* [12] versus *between subjects* in this work), data of all our participants (66) could be used for comparison of NO- versus OP-obfuscated code. [1] Two separate studies were conducted.

We opted for the *between subjects design* when comparing the performance of participants working on NO-obfuscated code with the performance of different participants working on OP-obfuscated code. For robust statistical analysis, between subjects design needs a higher number of participants. However, we let all participants first work on the clear code, because we decided to assess their level of expertise in program understanding in this way (see Section 3.3.3). This measurement of expertise should therefore be free from fatigue effects. This study design also lets us compare performance on non-obfuscated code with performance on obfuscated code for each participant (i.e., *within subjects*).

4.3.2 Groups and Tasks

The overall study design is presented in Table 4. The participants were randomly assigned to one of the four experimental groups. Each participant first worked on the clear code of one program, and then on the code of the other program obfuscated with NO or OP. For each program, the participant had to solve two tasks that are presented earlier in Table 2. The tasks were presented in the randomized order.

4.3.3 Procedure and Ethics

The study received approval by the data protection office of the Friedrich-Alexander-Universität Erlangen-Nürnberg. Participants worked under anonymous IDs and were informed at the beginning of their session about data collected during the experiment. We also explained that our goal is not to test their individual performance, but to understand in general how people work on various code comprehension tasks.

We conducted 14 sessions with 7 participants per session on average. Each session lasted 90 minutes, but the participants could leave earlier. In particular, if participants found the tasks too demanding, they could quit and were nevertheless fully paid. They received a 10 EUR gift voucher for participation. On average they worked for 47 minutes.

Each session started with a short presentation by the same researcher using the standardized set of slides. First, the purpose of software obfuscation was introduced, then the procedure was explained. The screenshots of the two programs were included, to make the participants familiar with

Group	1st Program (clear code)	2nd Program (obfuscated)
1	Race: *Rnd*(Box,Laps)	NO(Chat): *Rnd*(Messages,Users)
2	Race: *Rnd*(Box,Laps)	OP(Chat): *Rnd*(Messages,Users)
3	Chat: *Rnd*(Messages,Users)	NO(Race): *Rnd*(Box,Laps)
4	Chat: *Rnd*(Messages,Users)	OP(Race): *Rnd*(Box,Laps)

Table 4: Groups and tasks. Each user first worked on the clear code of one program, and then on the NO- or OP-obfuscated code of another program. *Rnd* denotes the randomization of task order within each program.

the programs. One or two additional researchers (depending on the number of the session participants) were in the lab to ensure the smooth execution of the experiment.

After the presentation, the participants logged into the virtual machine with their anonymized participant ID. There, they opened Firefox and started filling out the online survey. After answering the pre-study questionnaire, they were shown a password that they entered to unzip the zip-file with the program code. By entering the password, Eclipse was automatically set up with the corresponding source code (unobfuscated for the first program) according to the group the participants belonged to. Also, the logging of all events and timings in Eclipse started.

Back in the online survey, a description of the the first program was shown. On the next page of the survey the first task was presented and the solution had to be filled in. When the first task was successfully completed, the survey asked the post-task questions. Next, the second task was presented in the survey. After finishing this task, participants where asked to close Eclipse. By doing so, a log-file with all events in Eclipse was sent to our server. Participants then filled out post-task questions again. Furthermore, the post-program questions were asked. Then the password for the second program was shown and the same procedure was repeated for the second (obfuscated) program.

4.4 Data Analysis

Statistical analysis was performed using SPSS [29]. For all tests, a significance level of $\alpha = 0.05$ was employed.

4.4.1 Effect of Code Obfuscation

To compare code comprehension and code analysis behavior for clear and for obfuscated code, we used Wilcoxon signed-rank tests (within subjects design). To compare both obfuscation methods with each other, we used Mann-Whitney U tests (between subjects design). Non-parametric tests were used because the assumption of normal distribution was violated for most variables (as indicated by Shapiro-Wilk and Kolmogorov-Smirnov tests).

4.4.2 Impact of Experience

Experience was assessed with three measures: *Programming Experience*, *Study-relevant Experience*, and *Comprehension Skills* (Section 3.3.3). We first analyzed the five questions of *Study-relevant Experience*. With a factor analysis, we extracted two factors with eigenvalues larger than 1 (Kaiser Guttman criterion). These two factors explained 82% of the variance in the data. Table 5 shows the factor loadings after varimax rotation. Factor 1 summarizes experience with obfuscated code and debugging and Factor 2 encompasses experience with Java and Eclipse. Individual experience levels

	Factor 1	Factor 2
Code obfuscation	.921	
Debugging software	.798	
Java		.773
Eclipse for software development		.940
Eclipse for debugging		.925

Table 5: Factor loadings after varimax rotation. Values below 0.4 are omitted.

	Progr. Exp.	Obfusc. Exp.	Java Exp.	Compr. Skill[1]
Obfus.Exp.	0.648**			
Java Exp.	0.466**	0.298*		
Compr. Skill[1]	0.323**	0.411**	0.097	
Compr. Skill[2]	0.264*	0.326**	0.140	0.945**

Table 6: Correlations between experience indicators; [1]our measurement, [2]strict measurement (Ceccato et al.); *p<.05, **p<.01.

were computed by averaging across the respective questions. In summary, we consider four indicators of experience:

- *Programming Experience*: quality and type of code written so far;
- *Obfuscation Experience*: experience with obfuscation and debugging;
- *Java Experience*: experience with Java and using Eclispe;
- *Comprehension Skills*: efficiency in working on clear code.

The four indicators were moderately correlated with each other (see Table 6), indicating that they can be integrated to measure individual levels of experience.

On the basis of the four experience indicators, we divided participants into experience groups using a data-driven approach. We ran a cluster analysis, which tries to identify homogeneous groups of cases, such that observations in the same group are as similar as possible, and observations in different groups are as different as possible. A k-means cluster analysis was performed, setting the parameter k to the value 2 to extract two groups of experience. The final groups, "Beginners" ($N = 21$) and "Experienced" ($N = 45$), differed significantly in all four indicators, all F's$(64) > 5.952$, p's< 0.018 (see Table 7).

To assess the moderating effect of experience on code comprehension and code analysis behavior, mixed-model Analyses of Variance (ANOVA) were run with Obfucation (Clear vs. Obfuscated Code) as within subjects factor and Ex-

	Beginners $N = 21$	Experienced $N = 45$
Programming Exp.	1.42 ± 0.60	2.96 ± 0.52
Obfuscation Exp.	1.55 ± 0.44	2.99 ± 0.73
Java Exp.	2.30 ± 0.60	3.13 ± 0.80
Compr. Skill	0.07 ± 0.06	0.18 ± 0.20

Table 7: Description of the experience groups (Mean±SD).

perience (Beginners vs. Experienced) between subjects factor.[8] Effects of obfuscation (irrespective of experience) are reflected by the main effect Obfuscation. Similarly, effects of experience (irrespective of the type of code) is reflected in the main effect Experience. Whether experience moderates the obfuscation effect (i.e., whether beginners and experts differ in working with obfuscated code) is reflected by the interaction between Obfuscation and Experience. If the interaction was significant, we run post hoc t-tests in order to compare beginners and experienced programmers when working with obfuscated code.

4.4.3 *Effect Sizes and Statistical Power*

To assess the practical meaning of the empirical results, we calculated effect sizes. For Wilcoxon signed-rank tests and Mann-Whitney U tests, we report r. For ANOVAs we report partial eta-squared (η_p^2). For unpaired t-tests, we report Cohen's d. For paired t-tests, we report Cohen's d_z, which corrects the effect size for correlations in a within-subjects design. However, both Cohen's d and η_p^2 can be greater than 1, making an intuitive interpretation difficult. Therefore, we also report ω^2, which ranges between 0 and 1. It can be interpreted as the percentage of variance in the data that is explained by the experimental manipulation. For interpretation, we followed the convention provided by Cohen [30].

Interpretation	Cohen's d & d_z	r	η_p^2 and ω^2
no effect	< 0.20	<0.10	<0.01
small effect	0.20-0.50	0.10-0.30	0.01-0.06
medium effect	0.50-0.80	0.30-0.50	0.06-0.14
large effect	> 0.80	>0.50	>0.14

Table 8: Interpretation of effect sizes.

We assume that effects indicate practical relevance if they are of at least medium size (Table 8). A power analysis showed that we were able to detect such an effect in the population with a probability of $\beta = 0.80$ in a within subjects design with a sample of $N = 35$ participants (i.e., running a Wilcoxon test) and in a between subjects design with a sample of $N = 134$ (i.e., running a Mann-Whitney U test). Referring to the actual number of participants (Table 3), most of our tests (apart from Clear vs OP) are underpowered, meaning that we might have missed some effects due to small sample size.

[8] Although the assumption of normal distribution has been violated for most variables, to our knowledge, there is no valid non-parametric equivalent to a two-way ANOVA implemented in our analysis tool SPSS. For example, the Kruskal-Wallis test can be used as non-parametric equivalent to the one-way ANOVA. However, as we are interested in the interaction between two factors, i.e. Obfuscation and Experience, the test is not valid in our case.

		Evaluation method *This paper*	*Ceccato et al.*
Race	Box	78.8% (52/66)	78.8% (52/66)
	Laps	78.8% (52/66)	54.5% (36/66)
Chat	Messages	57.6% (38/66)	57.6% (38/66)
	Users	31.8% (21/66)	18.2% (12/66)

Table 9: Task correctness rates when evaluating the results with our evaluation method versus with the stricter rules by Ceccato et al. (Section 3.3.1).

5. RESULTS

We present our results and, if applicable, compare them with the findings of Ceccato et al. [12]. We start with descriptive results (Section 5.1), and then analyze differences between clear and obfuscated code with regard to code comprehension and analysis behavior (Sections 5.2, 5.3 and 5.4). The results of these evaluations are summarized in Table 10. Finally, we assess the moderating effect of experience (Section 5.5 and Table 11).

5.1 Descriptive Results

Correctness results are presented in Table 9. Each of the 66 participants worked on four tasks, two with clear and two with obfuscated code. Using our less strict evaluation of all 264 solutions (see Section 3.3.1), 163 were rated correct and the remaining 101 were false. Using the more strict evaluation by Ceccato et al., our participants scored 138 correct and 126 false answers. In both cases, "Chat: Users" was the most difficult task, and "Race: Box" the easiest one.

The fastest participant took 21 minutes, the slowest finished after 90 minutes. For the Chat program 90.9% and for the Race program 95.5% of the participants agreed or strongly agreed that the descriptions of the application was clear.

5.2 Name Overloading ($HC1_{NO}$ & $HB1_{NO}$)

Tasks with clear and obfuscated code were solved with similar correctness, $T(31) = 91.50$, $p = 0.373$, $z = -0.892$, $r = -0.113$ (Table 10). To show the same level of correctness with obfuscated code, participants needed significantly longer, $T(31) = 384.00$, $p = 0.008$, $z = 2.665$, $r = 0.338$. This speed-accuracy trade-off was reflected in a significant effect on efficiency, $T(31) = 104.00$, $p = 0.014$, $z = -2.454$, $r = -0.312$. Time needed to correctly solve a task, i.e., a successful attack, was significantly longer for obfuscated code, $T(21) = 181.00$, $p = 0.023$, $z = 2.277$, $r = 0.351$.

Using stricter correctness by Ceccato et al. [12], we also found no difference concerning the correctness of code comprehension between clear and obfuscated code, $T(31) = 67.50$, $p = 0.648$, $z = -0.456$, $r = -0.058$. The effect of NO on efficiency was significant, $T(31) = 106.00$, $p = 0.046$, $z = -1.994$, $r = -0.253$. The time to correctly solve tasks showed no difference between the groups, $T(20) = 152.00$, $p = 0.079$, $z = 1.755$, $r = 0.277$. However, our sample size was not sufficient to detect effects of medium size (Section 4.4.3), such that we might have missed some effects.

	Descriptive Results						Parameter-free tests		
	Clear		NO		OP		*Clear*	*Clear*	*NO*
Measurement	*Median*	*IQR*	*Median*	*IQR*	*Median*	*IQR*	*vs NO*	*vs OP*	*vs OP*
Correctness	2.000	1.000	2.000	1.000	1.000	1.000	91.50	80.50	496.00
Efficiency	0.130	0.161	0.096	0.079	0.090	0.057	104.00*	125.00**	571.00
Total time	13.163	11.404	18.154	9.028	15.986	11.551	384.00**	456.00*	444.00
Time correct	5.758	5.595	8.888	3.817	6.167	6.702	181.00*	166.00	216.00
Strict correctness as measured by Ceccato et al.:									
Correctness	1.000	1.000	1.000	1.000	1.000	1.000	67.50	77.00	492.00
Efficiency	0.111	0.105	0.063	0.082	0.082	0.052	106.00*	161.00	570.00
Time correct	5.439	6.079	8.874	3.817	5.951	7.012	152.00	156.00	207.00
Number of:									
File open commands	13.000	19.000	30.000	30.000	19.500	19.250	429.00**	344.50	307.00**
Advanced commands	0.000	3.000	1.000	11.000	1.000	4.250	121.50**	126.00*	479.50
Program executions	1.000	3.000	3.000	5.000	2.000	2.000	216.50	376.00**	478.00
Debugging mode	0.000	1.000	0.000	3.000	2.000	5.000	105.00**	138.00**	560.00
Time spent on:									
Program executions	0.383	1.400	1.317	4.467	1.025	1.504	265.50*	388.50*	534.00
Debugging mode	0.000	0.450	0.000	10.117	3.000	11.292	120.00**	131.00*	496.00
Code reading	10.500	9.533	13.683	9.633	10.700	8.004	280.00	323.00	491.50

Table 10: Descriptive results and parameter-free statistics (Wilcoxon & Mann-Whitney-U tests) comparing clear and obfuscated code; all times are in minutes; *p<.05, **p<.01.

Reduced efficiency and increased total time may be due to changes in code analysis behavior. Participants opened files more frequently, $T(31) = 429.00$, $p < 0.001$, $z = 3.548$, $r = 0.451$, used more advanced commands, $T(31) = 121.50$, $p = 0.006$, $z = 2.771$, $r = 0.352$, and more often the debugging mode, $T(31) = 105.00$, $p = 0.001$, $z = 3.311$, $r = 0.420$. Overall they spent more time with program executions, $T(31) = 265.50$, $p = 0.022$, $z = 2.286$, $r = 0.290$, and debugging, $T(31) = 120.00$, $p = 0.001$, $z = 3.408$, $r = 0.433$. The observed effects were of medium size.

In summary, obfuscating source code with NO significantly reduced the efficiency of code comprehension ($HC1_{NO}$). Participants changed their code analysis behavior ($HB1_{NO}$), i.e., they opened files more frequently, used more advanced commands, and the debugging mode. The observed effects were of medium size, indicating their practical importance. The behavior of participants corresponds to what can be expected when dealing with NO since the inverse transformation to NO (rename identifier) is an advanced command in Eclipse. Other increases can be explained by additional effort to understand the meaning of individual identifiers.

5.3 Opaque predicates ($HC1_{OP}$ & $HB1_{OP}$)

If the source code was obfuscated with opaque predicates, we observed similar effects (Table 10). Participants needed significantly longer to understand the code, $T(35) = 456.00$, $p = 0.021$, $z = 2.309$, $r = 0.276$, in order to reach about the same level of correctness, $T(35) = 80.50$, $p = 0.198$, $z = -1.286$, $r = -0.154$. This is reflected in reduced efficiency, $T(35) = 125.00$, $p = 0.005$, $z = -2.778$, $r = -0.332$. Concerning the time needed to correctly solve a task, i.e., a successful attack, no difference between clear and obfuscated source code was found, $T(22) = 166.00$, $p = 0.200$, $z = 1.282$, $r = 0.193$.

Again, using the stricter measurements by Ceccato et al. [12], participants reached about the same performance in terms of correctness, $T(35) = 77.00$, $p = 0.980$, $z = 0.025$, $r = 0.004$, and were marginally less efficient, $T(35) = 161.00$, $p = 0.054$, $z = -1.926$, $r = -0.326$.

The impact of code obfuscation was also visible in code analysis behavior. Participants more often used advanced commands, $T(35) = 126.00$, $p = 0.018$, $z = 2.359$, $r = 0.282$, executed the program more frequently, $T(35) = 376.00$, $p = 0.003$, $z = 2.979$, $r = 0.356$, executed code longer in total $T(35) = 388.50$, $p = 0.020$, $z = 2.328$, $r = 0.278$, used the debugging mode more often, $T(35) = 138.00$, $p = 0.003$, $z = 2.923$, $r = 0.349$, and spent more time debugging, $T(35) = 131.00$, $p = 0.010$, $z = 2.580$, $r = 0.308$.

In summary, obfuscating source code with OP significantly reduced the efficiency of code comprehension ($HC1_{OP}$). Participants changed their analysis behavior ($HB1_{OP}$), i.e., they used more advanced commands, executed the program more frequently, and used the debugging mode more often. The observed effects were of medium size, indicating their practical importance. Compared to the changes with NO, the differences in behavior between Clear and OP appear to be more random which can be interpreted as an unguided search for understanding.

5.4 Comparison of Obfuscation Methods (HC2 & HB2)

The previous analyses showed that both obfuscation methods, name overloading and opaque predicates, significantly reduced code comprehension performance. To achieve a similar level of comprehension, participants changed their behavior of code analysis. A direct comparison between both obfuscation methods indicates that code comprehension was hindered similarly, i.e., we found no differences in correctness, total time or efficiency. Also, the effect on time needed

to correctly solve a task, i.e., time for a successful attack, was small and non-significant, $U(50) = 216.00$, $p = 0.066$, $z = -1.839$, $r = -0.260$ (Table 10).

Concerning behavior, participants opened files significantly more frequently when the source code was obfuscated with NO than with OP, $U(66) = 307.00$, $p = 0.002$, $z = -3.027$, $r = -0.373$. This effect is of medium size.

In summary, both obfuscation methods reduced efficiency of code comprehension and led to similar behavior of the participants in almost all aspects. The number of file openings being higher in NO could be due to the fact that the structure of the code is not changed by the transformation and thus many aspects of semantics remain. The main effort is to deduce useful meanings of identifiers using static and dynamic analysis techniques. We would have expected a significant difference in using advanced commands for NO than OP due to the expected higher use of the advanced command "rename identifier", but the usage of this particular primitive does not appear to be different in the data. We note, however, that our sample size was too small for a between subjects comparison, such that we might have missed some effects (Section 4.4.3).

5.5 Impact of Experience (HC3 & HB3)

Here we assess whether experience moderates code comprehension and code analysis behavior. We performed ANOVAs with Obfuscation (clear vs. obfuscated) as within subjects factor and Experience (beginners vs. experienced) as between subjects factors (see Section 4.4.2). As the main effect of Obfuscation replicates the results reported before, we only report the main effect of Experience and the interaction between Obfuscation and Experience here. Descriptive results and inferential statistics are presented in Table 11.

5.5.1 General Effect of Experience

The difference between beginners and experienced programmers, irrespective of the type of code, is reflected in the main effect of Experience. Beginners and experienced participants spent about the same time to solve the tasks (15.8 minutes vs. 17.6 minutes), $F(1, 64) < 1$. As experienced participants solved about 1.4 tasks whereas beginners solved only 0.8 task in this time correctly, $F(1, 64) = 13.907$, $p = 0.001$, $\eta_p^2 = 0.18$, $\omega^2 = 0.16$, their efficiency was significantly higher, $F(1, 64) = 8.008$, $p = 0.006$, $\eta_p^2 = 0.10$, $\omega^2 = 0.10$. The effects were of medium size, explaining 10% to 16% of the variability in the data. The same results were observed for the strict correctness of Ceccato et al. [12].

Beginners and experienced programmers showed different code analysis behaviors. Experienced participants executed advanced commands ten times more often than beginners, $F(1, 64) = 11.157$, $p < 0.001$, $\eta_p^2 = 0.15$, $\omega^2 = 0.13$, and used the debugging mode eight times more often, $F(1, 64) = 11.252$, $p = 0.001$, $\eta_p^2 = 0.15$, $\omega^2 = 0.13$. This was also visible in the overall time they spent in debugging mode, $F(1, 64) = 4.531$, $p = 0.037$, $\eta_p^2 = 0.07$, $\omega^2 = 0.05$. The latter effect was small, the other effects were of medium size.

In summary, experienced programmers solved 36% more tasks correctly in about the same time as beginners, which was reflected in a higher efficiency. To analyze the code, experienced participants used advanced commands and the debugging mode more often, which is consistent with the expected behavior of experienced reverse engineers.

5.5.2 Experience as Moderator of Comprehension

Solving tasks with obfuscated code requires more time to keep the level of correctness, i.e., efficiency is lower. Moreover, working on obfuscated code requires a change in code analysis behavior (see Sections 5.2 and 5.3). We are now interested in whether beginners and experienced programmers show similar or different changes. Statistically, this effect is reflected by the interaction between Obfuscation and Experience in the ANOVAs.

With regard to code comprehension, experience moderated the obfuscation effect on efficiency significantly, $F(1, 64) = 4.385$, $p = 0.040$, $\eta_p^2 = 0.05$, $\omega^2 = 0.05$. Beginners' efficiency did not significantly change when working on obfuscated code (0.06 tasks per minute) compared to clear code (0.07 tasks per minute), $t(20) < 1$. In contrast, experienced programmers were significantly more efficient with clear code (0.19 tasks per minute) than with obfuscated code (0.08 tasks per minute), $t(44) = 3.499$, $p = 0.011$, $d_z = 0.68$. When working with obfuscated code, their efficiency almost dropped to those of beginners (i.e., 0.08 vs. 0.07 tasks per minute). This difference between beginners and experienced programmers was statistically not significant, $t(64) = 1.387$, $p = 0.174$, $d = 0.37$. One may argue that the statistical power was not sufficient to detect the effect. Indeed, the small effect size $d < 0.50$ indicates that there is probably a small effect in the population. That is, programming experience may have an advantage for comprehending obfuscated code efficiently but this advantage is probably only minor (see Figure 1). This issue needs further investigation with a more appropriate sample size.

This drop in efficiency for experienced participants, $F(1, 64) = 1.609$, $p = 0.209$, $\eta_p^2 = 0.03$, $\omega^2 = 0.01$, might be due to an increase in total time in order to keep a similar level of correctness, $F(1, 64) < 1$. That is, experienced programmers invested more time to keep a high level of correctness, whereas beginners did not. However, the effects were of quite small size and the sample size was insufficient to detect these effects statistically. Also, we were not able to replicate them using the strict measurements by Ceccato et al. [12], although they point into the same direction.

When working on obfuscated code compared to clear code, experienced programmers changed their code analysis behavior, whereas beginners did not (Figure 1). This moderating effect of experience occurred for the usage of advanced commands, $F(1, 64) = 5.321$, $p = 0.024$, $\eta_p^2 = 0.08$, $\omega^2 = 0.06$, and the usage of the debugging mode, $F(1, 64) = 7.615$, $p = 0.008$, $\eta_p^2 = 0.11$, $\omega^2 = 0.09$. All effects were of medium size, explaining 6% to 9% of the variability in the data.

In summary, code comprehension and analysis behavior of *beginners* was not much impacted by obfuscated code. As expected, *experienced programmers* were more efficient when working with clear code. However, code obfuscation impeded code comprehension. The efficiency dropped by 57% to those of beginners. In our view, this is the most interesting result of our study. To keep the higher level of correctness compared to beginners, experienced programmers invested more time to solve the tasks. They changed their code analysis strategies, i.e., they used more often advanced commands and the debugging mode. Overall they spent more time with reading the source code. It appears

	Descriptive Results								ANOVA		
	Beginner				Experienced						
	Clear		Obfucated		Clear		Obfuscated				
Measurement	Mean	SD	Mean	SD	Mean	SD	Mean	SD	*Obf.*	*Exp.*	*Obf.* × *Exp.*
Correctness	0.90	0.77	0.86	0.85	1.53	0.66	1.27	0.72	1.377	13.907**	0.669
Efficiency	0.066	0.059	0.057	0.060	0.185	0.202	0.079	0.059	6.185*	8.008**	4.385*
Total Time	14.4	7.6	17.3	11.8	13.9	9.1	21.3	10.8	8.917**	0.815	1.609
Time correct	8.4	3.9	10.4	8.9	6.3	5.2	9.5	5.9	3.183	0.768	0.170
Strict correctness as measured by Ceccato et al.:											
Correctness	0.62	0.74	0.71	0.78	1.27	0.62	1.18	0.68	0.001	16.213**	0.604
Efficiency	0.049	0.061	0.048	0.057	0.151	0.188	0.075	0.059	3.061	8.145**	2.925
Time correct	10.0	7.7	7.6	2.8	6.3	5.4	9.1	5.9	0.015	0.329	2.743
Number of:											
File open comm.	18.7	18.1	23.0	20.5	17.4	16.7	30.2	17.2	6.719*	0.752	1.641
Advanced comm.	0.3	0.8	0.4	0.8	2.3	3.4	5.4	7.1	5.662*	11.157**	5.321*
Program exec.	0.8	0.9	2.6	2.6	2.5	4.0	4.5	6.7	6.286*	3.225	0.021
Debugging mode	0.1	0.3	0.4	0.9	0.9	1.9	3.0	3.4	13.368**	11.252**	7.615**
Time spent on:											
Program exec.	0.8	1.6	3.2	4.4	1.5	2.5	3.4	7.7	5.800*	0.169	0.081
Debugging mode	0.1	0.2	3.1	9.2	1.7	4.5	7.0	8.4	12.793**	4.531*	1.019
Code reading	14.0	8.7	12.1	8.0	11.0	8.1	13.8	9.1	0.064	0.157	2.110

Table 11: Descriptive results and inferential statistics comparing clear and obfuscated code for beginners and experienced programmers; all times are in minutes; *p<.05, **p<.01.

that classical programming experience does not help much in comprehending obfuscated source code.

5.5.3 Exploration: Areas of Experience

The previous analysis shows that no evidence was found that experienced programmers, who were much more efficient with clear code than beginners, differ from the latter when the source code was obfuscated. In the following we explore whether experience in a particular area may prevent from the drop in efficiency.

We correlated the level of experience (in one of the four areas of experience we had measured, see Sectionsec:analysis-experience) with the efficiency in working with obfuscated code (*Pearson* correlation). A positive correlation indicates that more experienced participants were able to keep a higher level of efficiency. Maybe not surprisingly, this was indeed the case if participants had *experience with obfuscated code and debugging* before, $r = 0.43$, $p < 0.001$. Neither programming experience in general, $r = 0.14$, $p = 0.271$, experience with Java and Eclipse, $r = 0.08$, $p = 0.544$, nor comprehension skills (measured as efficiency in working on clear code), $r = 0.16$, $p = 0.191$, did help. We conclude that reverse engineering needs special training in obfuscation techniques.

6. DISCUSSION

Before this study, we knew that obfuscation impeded program comprehension [12]. We were able to reproduce these findings for the same obfuscation methods, NO and OP, that were previously studied. We also obtained original results by studying the reverse engineering behavior. As might be expected, we found many significant differences in behavior between clear and obfuscated versions. These differences appeared to be more intentional for NO than OP. Participants appeared to have a clear strategy in countering NO but were still inhibited severely regarding efficiency. Given OP-obfuscated code, the analysis behavior appeared to be more random both in numbers of commands and time spent on different activities. With such a behavior, a decrease in efficiency is an understandable consequence.

Overall, the different behaviors for NO and OP are a first empirical support of the taxonomy of Collberg et al. [6] who distinguished obfuscating transformations regarding resilience and potency. For NO we found significant decreases in efficiency despite clear and understandable adaptions in behavior by participants. Such a strategic behavior change was not observable with OP. NO therefore appears to belong to the class of potent obfuscation techniques, increasing the "obscurity" for the human reverse engineers.

Furthermore, obfuscation seems to "reduce experience", i.e., the effect of software engineering experience on the success of program comprehension is much lower for obfuscated code than for unobfuscated code. This insight is important since it indicates that code comprehension in the realm of obfuscated software may be different from comprehension of traditional programs. We conjecture that comprehension strategies follow a two-step approach: in the first step the particular obfuscation method is identified; in a second step, an inverse transformation is attempted. Such a strategy can, however, only be applied if reverse engineers have (1) an understanding of different obfuscation techniques, and (2) the ability to inverse the obfuscation using ingenuity and/or tools. This is consistent with the findings of our experiment where understanding of obfuscation methods was more helpful than general programming experience.

7. LIMITATIONS

Because the analyzed programs and tasks could be of different difficulty, we used counterbalancing to mitigate this concern. We did not counterbalance clear and obfuscated code

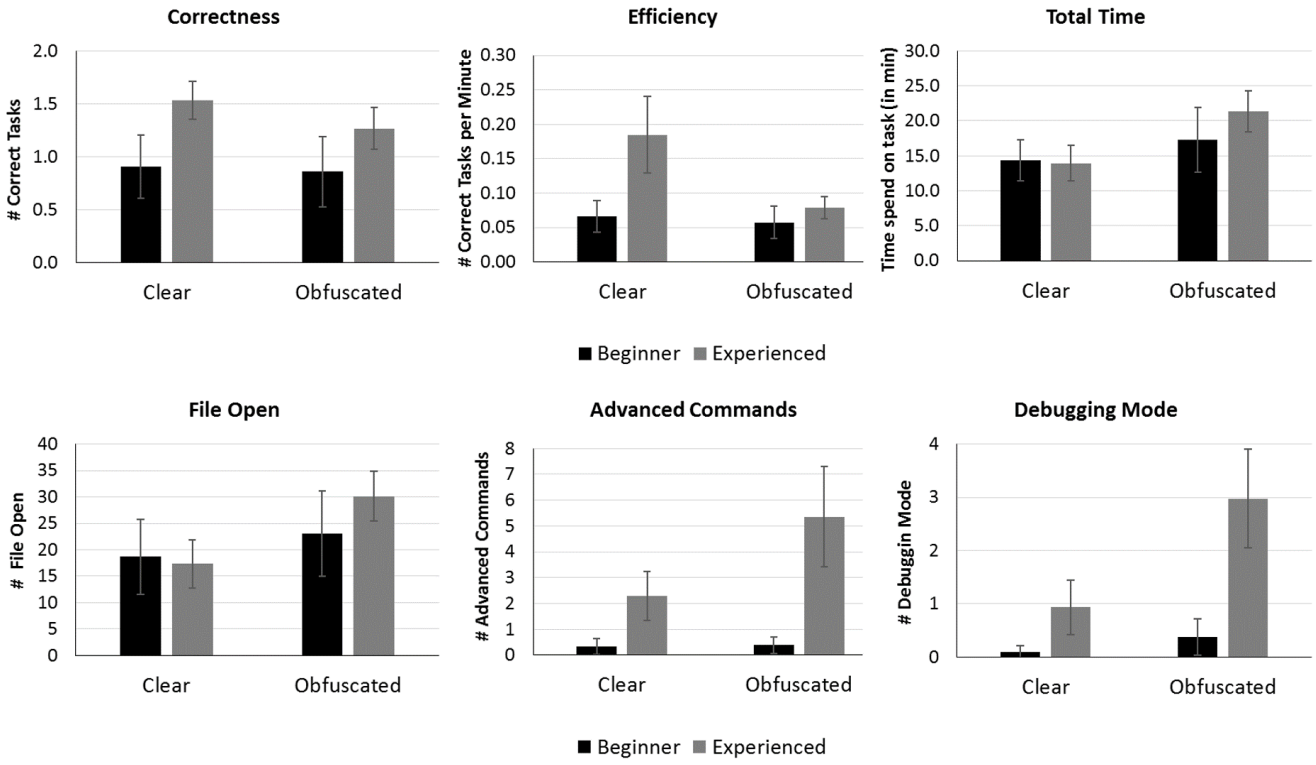

Figure 1: On clear code, experienced users were more efficient than beginners. This advantage of experience disappeared when the code was obfuscated. Experienced users invested more time to keep their level of correctness, but beginners did not. Beginners did not change their code analysis behavior when working on obfuscated code. Experienced users opened files more frequently and used advance commands and the debugging mode more often than when working on clear code. (Error bars indicate confidence intervals.)

tasks, as the participants worked on the clear code first. This was necessary for precise assessment of their comprehension skills. Therefore, learning effects may have positively influenced performance on obfuscated code, such that effects of obfuscation on code comprehension and behavior may actually be stronger than we found. In contrast, fatigue effects could have negatively influenced performance on obfuscated code. To counter this limitation, we analyzed only the data of participants who indicated that they had enough time to perform all tasks.

The sampling of experience was performed post hoc by placing participants into groups and not as a planned sampling based on experience. The representativeness of the sample (students) is limited, although the study by Acar et al. [18] provided evidence that experience may be a more important indicator of expertise than student status. A similar study with professionals using their own analysis equipment and a more realistic scenario (e.g., malware analysis) would be desirable, but would be hard to pursue given the scarcity of obfuscation analysis resources in the professional market.

8. CONCLUSIONS

In this work we measured effects of source code obfuscation on program comprehension skills of reverse engineers by means of a controlled experiment with 66 participants. We successfully replicated results by Ceccato et al. [12] that obfuscation techniques have a significantly negative effect on program comprehension. We also showed that the obfuscation methods NO and OP lead to significantly different analysis behavior. The differences provided insight into the relative strength of NO which withstood reverse engineering efforts, although it was clearly identifiable and the de-obfuscation tool (rename identifer) was available. This supports the distinction between resilience and potency of obfuscating transformations as defined by Collberg et al. [6] more than 20 years ago.

Future research should focus more specifically on the behavior of humans when facing obfuscated code. Do they follow a two-step approach as conjectured above? What if an unknown obfuscation technique or the combination of several is used? How do performance results change for professional malware analysts, or when deobfuscation tools are used?

Acknowledgment

This work was supported by the "Bavarian State Ministry of Education, Science and the Arts" as part of the FORSEC research association. We thank the anonymous reviewers for their helpful comments, and we are indebted to Mariano Ceccato, Brian Glass, Tilo Müller, Yan Zhuang and our shepherd Joseph Bonneau for their invaluable support.

9. REFERENCES

[1] J. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. W. Smith, and S. Weingart, "Building the IBM 4758 Secure Coprocessor," *IEEE Computer*, vol. 34, no. 10, pp. 57–66, 2001.

[2] U. Piazzalunga, P. Salvaneschi, F. Balducci, P. Jacomuzzi, and C. Moroncelli, "Security strength measurement for dongle-protected software," *IEEE Security & Privacy*, vol. 5, no. 6, pp. 32–40, 2007.

[3] T. Alves and D. Felton, "TrustZone: Integrated hardware and software security," ARM, Tech. Rep., Jul. 2004.

[4] O. Dvir, M. Herlihy, and N. Shavit, "Virtual leashing: Creating a computational foundation for software protection," *J. Parallel Distrib. Comput.*, vol. 66, no. 9, pp. 1233–1240, 2006. [Online]. Available: https://doi.org/10.1016/j.jpdc.2006.04.013

[5] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," in *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO '01. London, UK, UK: Springer-Verlag, 2001, pp. 1–18. [Online]. Available: http://dl.acm.org/citation.cfm?id=646766.704152

[6] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Technical Report 148, Department of Computer Science, University of Auckland, Jul. 1997. [Online]. Available: http://citeseer.ist.psu.edu/collberg97taxonomy.html

[7] A. Capiluppi, P. Falcarin, and C. Boldyreff, "Code defactoring: Evaluating the effectiveness of java obfuscations," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*, Oct 2012, pp. 71–80.

[8] M.-A. Storey, "Theories, tools and research methods in program comprehension: past, present and future," *Software Quality Journal*, vol. 14, no. 3, pp. 187–208, 2006.

[9] M. D. Penta, R. K. Stirewalt, and E. Kraemer, "Designing your next empirical study on program comprehension," in *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on*. IEEE, 2007, pp. 281–285.

[10] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *Software Engineering, IEEE Transactions on*, vol. 35, no. 5, pp. 684–702, 2009.

[11] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[12] M. Ceccato, M. Di Penta, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella, "A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques," *Empirical Software Engineering*, vol. 19, no. 4, pp. 1040–1074, 2014.

[13] A. Viticchié, L. Regano, M. Torchiano, C. Basile, M. Ceccato, P. Tonella, and R. Tiella, "Assessment of source code obfuscation techniques," in *Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on*. IEEE, 2016, pp. 11–20.

[14] S. Garfinkel and H. R. Lipford, "Usable security: History, themes, and challenges," *Synthesis Lectures on Information Security, Privacy, and Trust*, vol. 5, no. 2, pp. 1–124, 2014.

[15] D. Oliveira, M. Rosenthal, N. Morin, K.-C. Yeh, J. Cappos, and Y. Zhuang, "It's the psychology stupid: how heuristics explain software vulnerabilities and how priming can illuminate developer's blind spots," in *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 2014, pp. 296–305.

[16] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky, "You get where you're looking for: The impact of information sources on code security," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 289–305.

[17] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky, "Comparing the usability of cryptographic apis," in *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 2017, pp. 154–171.

[18] Y. Acar, C. Stransky, D. Wermke, M. L. Mazurek, and S. Fahl, "Security developer studies with github users: Exploring a convenience sample," in *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*. Santa Clara, CA: USENIX Association, 2017, pp. 81–95. [Online]. Available: https://www.usenix.org/conference/soups2017/technical-sessions/presentation/acar

[19] C. Collberg and J. Nagra, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*, 1st ed. Addison-Wesley Professional, 2009.

[20] M. Ceccato, M. Di Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella, "Towards experimental evaluation of code obfuscation techniques," in *Proceedings of the 4th ACM workshop on Quality of protection*. ACM, 2008, pp. 39–46.

[21] M. Ceccato, M. D. Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella, "The effectiveness of source code obfuscation: an experimental assessment," in *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*. IEEE, 2009, pp. 178–187.

[22] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[23] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, 2015, pp. 674–691. [Online]. Available: https://doi.org/10.1109/SP.2015.47

[24] Y. Yoon and B. A. Myers, "Capturing and analyzing low-level events from the code editor," in *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*, ser. PLATEAU '11. New York, NY, USA: ACM, 2011, pp. 25–30. [Online]. Available: http://doi.acm.org/10.1145/2089155.2089163

[25] C. Collberg, G. Myles, and A. Huntwork, "Sandmark–a tool for software protection research," *IEEE security & privacy*, no. 4, pp. 40–49, 2003.

[26] P. Kouznetsov, "Jad-the fast java decompiler," *URL: http://www. kpdus. com/jad. html*, 2006.

[27] A. Field and G. Hole, *How to design and report experiments*. Sage, 2002.

[28] G. Leroy, *Designing User Studies in Informatics*. Springer Science & Business Media, 2011.

[29] A. Field, *Discovering statistics using IBM SPSS statistics - 4th edition*. Sage, 2013.

[30] J. Cohen, *Statistical power analysis for the behavioral sciences . Hilsdale*. NJ: Lawrence Earlbaum Associates, 1988.

APPENDIX

A. EXAMPLES OF OBFUSCATED CODE

We illustrate the code obfuscation techniques by showing excerpts of code from our study in the Race program. Listing 1 shows the definition of the method `changeSpeed` from the file `MovingCarModel.java`, which changes the speed by a certain value (given as a parameter) depending on whether the car still has fuel (a value stored in the variable `gas`).

Listing 1: Clear code from Race `MovingCarModel.java`

```
1  public void changeSpeed(int i)
2  {
3      if(started)
4          if(gas == 0)
5          {
6              speed += i;
7              if(speed > maxSpeed / 10)
8                  speed = maxSpeed / 10;
9              else
10             if(speed < minSpeed / 10)
11                 speed = minSpeed / 10;
12         } else
13         {
14             speed += i;
15             if(speed > maxSpeed)
16                 speed = maxSpeed;
17             else
18             if(speed < minSpeed)
19                 speed = minSpeed;
20         }
21 }
```

Listing 2 shows the same code after applying *name overloading* where all identifiers have been renamed to some arbitrary values that have nothing to do with program semantics. The comparison between Listings 1 and 2 shows that apart from changing names of identifiers, the code stays structurally the same (e.g. lines 6 and 27 correspond).

Listing 3 shows the code from Listing 1 after introducing *opaque predicates*. In the given version of Sandmark, opaque predicates are generated using queries to a tree data structure which is manipulated in randomly looking ways. Predicates are then used to insert dead code that uses valid identifiers in random ways (see for example lines 45–47). To determine the truth value of a predicate (and therefore to eliminate dead code), an analyst has to first understand the way in which the tree was changed. The example shows that while all original code is maintained (e.g. line 6 corresponds to line 57), opaque predicates can be used to considerably complicate a program's control flow.

Listing 2: Code from Listing 1 obfuscated with NO

```
22 public void __m1(int i)
23 {
24     if(__f22)
25         if(__f19 == 0)
26         {
27             __f5 += i;
28             if(__f5 > __f6 / 10)
29                 __f5 = __f6 / 10;
30             else
31             if(__f5 < __f7 / 10)
32                 __f5 = __f7 / 10;
33         } else
34         {
35             __f5 += i;
36             if(__f5 > __f6)
37                 __f5 = __f6;
38             else
39             if(__f5 < __f7)
40                 __f5 = __f7;
41         }
42 }
```

Listing 3: Code from Listing 1 obfuscated with OP

```
43 public void changeSpeed(int i) {
44   if (Node.getI() != Node.getH()) {
45     lastFuel = (0L + time2) - (long) lap;
46     started = lastFuel == 0L;
47     Node.getF().setLeft(Node.getH().getLeft());
48   } else {
49     Node.getG().getLeft().swap(
50         Node.getG().getRight());
51     if (started)
52       if (Node.getI() == Node.getH()) {
53         if (gas == 0) {
54           if (Node.getF() == Node.getG()) {
55             Node.getF().setLeft(
56                 Node.getI().getRight());
57             speed += i;
58           } else {
59             [...]
60           }
61           if (Node.getI() != Node.getH()) {
62             lap = 1 + maxSpeed / status;
63             time += maxSpeed;
64             Node.getH().setLeft(
65                 Node.getH().getLeft());
66           } else {
67             Node.getF().getRight().swap(
68                 Node.getF().getRight());
69             if (speed > maxSpeed / 10) {
70               if (Node.getF() != Node.getG()) {
71                 Node.getF().getLeft().swap(
72                     Node.getH().getLeft());
73                 track = track;
74               } else {
75                 speed = maxSpeed / 10;
76                 Node.getF().setLeft(
77                     Node.getF().getLeft());
78               }
79   [...]
80 }
```

B. THE ONLINE SURVEY

In this section we present the full survey used in the study. This survey was slightly adapted from the materials of Ceccato et al. [12] to reflect the fact that we measured times of task completion and programming behavior, whereas Ceccato et al. asked their participants to note down their starting and finishing times, and to estimate the percentage of time they spent on reading and running the code, and executing the code in debugging mode.

B.1 Pre-Test Questions

At first the participants filled out a pre-test questionnaire in the online survey.

Q1. What is your position?

- ( ) Bachelor student
- ( ) Master student
- ( ) Diploma student
- ( ) PhD student
- ( ) Post Doc
- ( ) Professor
- ( ) Other:

Q2. What is your study subject? (this question was only displayed for participants who are either Bachelor, Master or Diploma student)

Q3. At which department are you working? (this question was only displayed for participants who are either PhD student, Post Doc or Professor)

Q4. How old are you?

Q5. How would you describe the quality and the type of the code you wrote so far?

- ( ) Few and small programs (e.g., course exercises)
- ( ) Many small programs
- ( ) Small programs and few (1 or 2) big programs (e.g., thesis and projects)
- ( ) Big programs

Q6. Have you ever worked as computer programmer?

- ( ) No
- ( ) Yes, part-time
- ( ) Yes, full-time

Q7. Do you have high or very high experience in any programming language(s)? If yes, please name them:

Q8. Did you participate in a Software Reverse Engineering or Hacking Lab Course?

- ( ) Yes
- ( ) No
- ( ) Other:

What is your experience/knowledge in... (5-point Likert scale from "very low" to "very high")

Q9. code obfuscation?

Q10. Java?

Q11. the usage of Eclipse for software development?

Q12. debugging software?

Q13. the usage of Eclipse for debugging software?

Before being able to work on the programs a password was displayed in the survey to gain access to the directory of the source files of the first program. This was done in order to prevent participants from analyzing the source code before the tasks to solve were presented.

B.2 Program Descriptions

After the participants got access to the the source files for a program, a short description about the program's general usage was displayed in the survey.

Race program

CarRace is a network game that allows two players run a car race.

The player that first completes the total number of laps wins the race. Use the arrow keys to control the car (your car is the green one). Keep "up" and "down" keys pressed to accelerate and brake. Press "right" and "left" arrows to turn right and left.

The car constantly consumes fuel, when the car runs out of fuel the speed drops. In order to avoid this case the players should stop at the box to refuel. The number of completed laps and the fuel level is displayed on the upper part of the window.

Chat program

ChatClient is a network application that allows people to have text based conversation through the network. Conversations can be public or private, depending on how they are initiated.

The application shows on the right a list of available rooms. When the application starts, the "default" room is accessed. It is a public room where all the users are participating. In order to access another room (e.g., Room 1) the name of the room must be clicked from the "Available Rooms" list, a new tab will be visualized. All the messages sent to a conversation within a room are received to all the users registered to that room.

A private conversation (only two users) can be initiated by clicking the name of a user from the "Online Users" list.

B.3 Questions after each task

After the completing the Pre-Test Questions the participants worked on the tasks as specified in the main part of the paper in Table 2. The tasks were presented in random order. After each task the following question had to be answered by all participants.

Q14. Did you have enough time to solve this task?

- ( ) Yes
- ( ) No

Only if the participant had enough time, the next questions were displayed and had to be answered using a 5-point Likert scale from "strongly agree" to "strongly disagree".

Q15. I had enough time to perform the tasks

Q16. The description of the task was perfectly clear to me

Q17. I experienced no difficulty in the identification of the segment of code relevant for the task

Q18. The debugging environment is useful to execute the task

Q19. I found the Eclipse Refactor facility useful for this task

Q20. For this task I spent a lot of time reading the code

Q21. For this task I spent a lot of time running the code

B.4 Questions after completing both tasks for a program

After both tasks and the corresponding questions about them were answered, questions regarding the programs itself were posed. Again, a 5-point Likert scale from "strongly agree" to "strongly disagree" had to be used.

Q22. The description of the application was clear

Q23. I experienced no difficulty in understanding the program

Q24. Running the code was useful to understand the code

The completion of these answers for the first program led to the password for getting access to the second program being displayed.

B.5 Post-Test Questions

After the questions about the second program were answered, some post-test questions had to be answered using a 5-point Likert scale from "strongly agree" to "strongly disagree".

Q25. I experienced no difficulty in using the development environment (Eclipse)

Q26. I experienced no difficulty in using the Eclipse debugger

The experiments were conducted over the course of two semesters. For the second semester we added two questions at the end of the survey in which the participants could indicate if they worked on similar code before. These questions were added after an additional analysis of the literature on code comprehension, where the so-called *domain experience* emerged as an additional performance factor [8].

Q27. Have you ever programmed any kind of program which was in your personal opinion similar to the chat program? If yes, please specify

Q28. Have you ever programmed any kind of program which was in your personal opinion similar to the race game?