RE-Mind: a First Look Inside the Mind of a Reverse Engineer

Alessandro Mantovani and Simone Aonzo, *EURECOM;*
Yanick Fratantonio, *Cisco Talos;* Davide Balzarotti, *EURECOM*

https://www.usenix.org/conference/usenixsecurity22/presentation/mantovani

This paper is included in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Open access to the Proceedings of the 31st USENIX Security Symposium is sponsored by USENIX.

RE-Mind: a First Look Inside the Mind of a Reverse Engineer

Alessandro Mantovani
EURECOM

Simone Aonzo
EURECOM

Yanick Fratantonio
Cisco Talos

Davide Balzarotti
EURECOM

Abstract

When a human activity requires a lot of expertise and very specialized cognitive skills that are poorly understood by the general population, it is often considered 'an art.' Different activities in the security domain have fallen in this category, such as exploitation, hacking, and the main focus of this paper: binary reverse engineering (RE).

However, while experts in many areas (ranging from chess players to computer programmers) have been studied by scientists to understand their mental models and capture what is special about their behavior, the 'art' of understanding binary code and solving reverse engineering puzzles remains to date a black box.

In this paper, we present a measurement of the different strategies adopted by expert and beginner reverse engineers while approaching the analysis of x86 (dis)assembly code, a typical static RE task. We do that by performing an exploratory analysis of data collected over 16,325 minutes of RE activity of two unknown binaries from 72 participants with different experience levels: 39 novices and 33 experts.

1 Introduction

Researchers of different fields have studied, from a cognitive perspective, how humans perform several relevant activities with the goal of better understanding, improving, or automating field-related processes. For instance, in the area of computer science, many experiments have been conducted to study the mechanisms behind human's decisions in several tasks, ranging from program comprehension [30, 43] to human-computer interaction [7, 27], and from problem solving [12, 33] to computer security [34, 54]. The crossover between the human mind and computer science has also resulted in the creation, and in the recent rapid evolution, of the field of artificial intelligence.

On the one hand, fully autonomous systems have already replaced humans in several security-related tasks including, among the others, host and network-based attack detection [37, 48, 59], malware classification [35, 41, 50] and phishing detection [4, 25, 36]. On the other hand, other areas are still mostly human-driven. For instance, binary *reverse engineering* (RE) is still performed entirely by highly skilled security experts. Machines play an essential role in the process in the form of tools to unpack, disassemble, emulate, and perform binary similarity. However, humans are still responsible for "understanding" the code, which is the main goal in problems such as malware analysis or vulnerability discovery. This requires considerable expertise, together with a long and tedious manual effort. Unfortunately, the limited number of expert reverse engineers in the world is insufficient to cope with our society's security needs and the continuous growth in the amount of released software. The recent DARPA Cyber Grand Challenge (CGC) drove progress in computers' ability to reason about program binaries autonomously and discover vulnerabilities. However, these programs are still far from being able to compete against RE experts[1].

To overcome this problem, we believe it is fundamental to first understand *how humans approach and solve static RE tasks*. The comprehension of the most effective RE strategies used by expert humans can drive further research in the development of automated approaches, but it can also help design tailored training programs that can increase the number and the effectiveness of our experts.

Let us use a simple analogy to introduce the motivation for our work. When a professional chess player decides her next move, she has hundreds of millions of possible combinations to evaluate. However, previous research on the human brain of chess players has shown that this is not the way she reasons. Her brain can instead recognize patterns and naturally focus only on a handful of possible "good" moves. Now think about an expert reverse engineer. Similarly to a chess master, she also does not "evaluate" every single line of assembly code in a program, but she just skims through the code, focusing only on those critical parts to understand the code's logic. We believe

[1] The 2016 DEF CON CTF final put the best DARPA cyber-reasoning system (Mayhem [17]) against human teams. The supercomputer ended up in the last position [1] (even on simplified challenges explicitly written to accommodate the limited architecture supported by the machine).

that her primary skill is not to read faster every single basic block, but instead that she does not waste time reversing the ones that are not important for her task. In other words, she can see patterns where others can only see endless lines of code.

Sadly, today we do not know whether this hypothesis or any other hypothesis about how RE experts think is correct. At the 2020 Usenix Security conference, Votipka et al. [53] presented the first human study about RE. This work inspired our follow-up study, where the main focus is restricted to *static RE* (from now on used alternatively with RE) from the perspective of assembly code comprehension. Our goal is to investigate a set of hypothesis by means of quantitative measurements and statistical tests conducted on fine-grained recordings of real RE tasks.

To accomplish that, we try to answer the following research questions:

- What do experts do differently from novices?
- Do experts/novices share particular strategies to explore binary code?
- How are these strategies linked to the binary code elements (e.g., functions, basic blocks)?
- Is any particular strategy correlated with better RE performances?

To recruit a sufficient number of geographically-distributed participants, we designed an online platform that mimics the UI of traditional interactive disassemblers. We then used our platform to record the fine-grained behavior of 72 reversers while they solved two different reverse engineering exercises. In total, we collected 272 hours of binary reverse engineering activity, which we then analyzed to identify patterns and strategies that we can use to model the 'experience' of a reverser.

The results of our experiments allowed us to confirm that experts indeed visit less basic blocks than beginners, and they are also able to dismiss on average 22% of the blocks they visit in under two seconds. While novices tend to re-visit the same parts of the code multiple times, experts gain more information during their first visit. We also identified several exploration strategies, both at the basic block and the function level, that seems positively correlated with experience. For example, beginners are more likely to explore a binary 'horizontally,' while more skilled reversers are more likely to proceed in a vertical way.

These are only a few examples of the many features we investigate in this paper to characterize the static reverse engineering process. We believe that this fascinating area of system security, where human experience is highly regarded but little understood, can help our community to better understand the mechanics behind the cognitive aspects of reverse engineering.

2 Related Work

To the best of our knowledge, only four studies have been conducted so far on human behaviors in the context of reverse engineering [11, 15, 46, 53].

One of the first studies was conducted by Sutherland et al. [46] in 2006 to demonstrate that the education/technical knowledge and the ability to reverse engineer simple binary files are positively correlated. In 2012, Bryant [11] performed a semi-structured interview with the addition of in-place observations during the RE sessions to investigate four experts approaching typical RE scenarios, such as breaking the protection scheme of a toy binary. The outcome is a precise observation of the skills, mental flows, and knowledge-based techniques that the subjects exhibit while reversing a binary. Interestingly this work tries to be the bridge between the source code comprehension community and the RE one, by studying how reverse engineers make use of assembly patterns.

In 2018, Claire et al. [15] proposed RevEngE, a framework to monitor reverse engineering from several points of view. The framework is based on an instrumented virtual machine that registers events such as spawning a new process, focusing on a window and mouse clicks. The goal of this work was to describe a system that acts as a base for an observational study but the paper does not contain any measurements about how reverse engineers perform their activities. Even though the authors did not perform any experiments, the study still deserves a special attention because it proposes an approach which is suitable to perform a quantitative study of RE.

Finally, the recent work of Votipka et al. [53] is what we can consider as the first human study about RE, focusing on what high-level process reverse engineers follow and what technical approaches they adopt. The authors' goal was to improve the design of RE tools to make them more usable and intuitive. However, due to the lack of prior work outlining REs' processes and no theoretical basis for building quantitative assessments, the authors also performed a number of semi-structured interviews in which 16 participants recalled anecdotes of a binary they had reverse engineered in the past. This provides technical details about their strategy and experience, including when they switched from a tool to another, which hypothesis they formulated, and which type of documentation they consulted.

If the literature covering RE is scarce, a vast amount of work has been performed instead in the program comprehension field. Indeed, RE can be seen as a program comprehension problem applied to assembly code, with the goal of recovering the high-level abstractions needed to understand the program logic. For this reason, we collect here the most critical human studies related to program understanding. One of the leading research directions in program comprehension shows that programmers adopt non-linear ways to interpret source code, reasoning at a level of abstraction higher than the code itself [5, 6, 10, 29, 30, 43]. A well-known model about these high-level representations is what researchers refer to as *beacons*: beacons are patterns that experienced programmers can recognize when reading the source code [22, 28, 40]. The utility of beacons is mainly related to assessing some hypotheses that developers do about some unknown parts of the program, such as when they need to maintain some code base, as described

by Littman et al. [32]. Alternatively, Gugerty [21] argues that developers can use debuggers to verify some behaviors within the source code they are analyzing (e.g., by checking whether a variable contains the expected value at some point of the execution). It is also worth mentioning that some of these papers study program comprehension by performing a comparison between *experts* and *novices* [20, 21, 56]. We believe this to be a critical factor in understanding the impact of the experience, and this methodology served as inspiration for the experiments we present in this paper.

Finally, few studies have investigated the usability of RE tools. For instance, researchers have looked at improving the usability of decompilers [24, 57], showing that better variable naming and a reduced number of GOTOs affected positively the readability of the pseudocode. In the context of vulnerability discovery, Do et al. [18] proposed a static analysis framework that allows the developers to write code and run in parallel the static analyzer to help programmers to better manage the large number of alerts generated by the tool. In 2017, Shoshitaishvili et al. [45] showed that the communication between a fuzzing engine and non-skilled reverse engineers can increase the rate of discovered vulnerabilities by taking advantage of human intuition.

3 Scope of the study

Reverse Engineering is a broad topic that covers several different activities. Therefore, in this section, we emphasize what aspects we studied in our work and the actual focus of the paper.

In system security, we refer to binary RE as the activity by which a human, the Reverse Engineer, analyzes an executable file, either in whole or in part, to recover design and implementation information useful to understand the program functionalities. Depending on the context (e.g., malware analysis, vulnerability discovery, firmware analysis), the output of a reverse engineering analysis can be different. However in all cases the analyst is interested in reconstructing the logic of the program and in understanding which conditions must be met to reach a specific location in the code — which can be related to a bug or to a suspicious behavior in the case of malicious files [58].

Independently from its goal, the RE process usually involves different phases, and different tools are used to inspect the program and collect the required information. Some popular frameworks that support the analyst in this complicated task are interactive disassemblers, such as IDA Pro [3] and Ghidra [2]. These tools combine multiple functionalities (e.g., a disassembler, a decompiler, a debugger) in an interconnected and interactive user interface, which allows the analyst to inspect an enriched representation of the binary code. Reverse engineers often rely on a combination of both static analysis and dynamic analysis. The former consists of detailed observation of the binary components (e.g., functions, basic blocks, assembly instructions) to reconstruct the program's behavior without executing it; the latter relies instead on a step-by-step observation of the way a binary interacts with the memory and the operating system at runtime. In this study, we focus our investigation on the core activity that is part of any binary RE process: the static code understanding as presented by interactive disassemblers.

Although this activity represents only one portion of the RE process, we believe it deserves a special attention for several reasons. First, it is particularly interesting as the low-level nature of the Assembly language forces the human mind to make an additional effort when reading the instructions. In fact, the reverser needs to understand the effects of what she reads on the machine that will execute the code as well as mentally reconstruct high-level patterns (such as loops and branch conditions) and data types.

Moreover, this approach mirrors the initial studies of the program comprehension community, where various authors initially focused on how users read source code rather than directly embedding debuggers in their experiments [5, 6, 10, 29, 30]. For these reasons, we decided not to include any decompilers/debuggers in our pipeline, focusing instead on an in-depth analysis of the static assembly code comprehension process.

4 Methodology

To conduct a detailed investigation of how humans perform a RE task, we needed to replace the interview format adopted by previous studies with a fine-grained observation of subjects' actual behavior when requested to perform different tasks related to binary reverse engineering.

While the required data could be easily collected in a lab, for instance, by using eye-tracking equipment to monitor the participant behavior [9,16,38,49], this approach would introduce several problems. First of all, skilled reverse engineers are rare and remotely-accessible experiments are required to collect enough participants with different backgrounds. Second, even simple RE exercises require hours of concentration, which is difficult to achieve while under observation in a lab (especially when the candidate needs to keep her head stable to allow for proper monitoring). Therefore, we opted for implementing a web-based platform specifically designed to conduct our experiments.

The platform needs to be capable of extracting many low-level metrics, such as how much time a person spends looking at each basic block, how she explores and navigates the binary program, and how she annotates and manipulate the assembly code (e.g., by renaming functions and variables) along the way. Moreover, the interface needs to closely resemble the interface of existing reverse engineering tools (such as IDA Pro, Ghidra, and Binary Ninja) to let the users interact with a familiar environment. Finally, the system should incorporate special techniques (such as Restricted Focus Viewer [26] to blur basic blocks that are not currently selected) and a variety of instrumentations to collect a rich set of *raw* low-level information.

The low-level metrics extracted by our online platform act as basic blocks for the subsequent analyses and characterizations. In this second phase, we manually reviewed the collected data

Figure 1: Part of the UI of our reverse engineering framework in the code navigation mode showing on the left the functions' list and on the right the CFG of the selected function

to identify high-level skills starting from the low-level metrics. In this respect, a significant challenge is that we did not know a priori which skills were more important than others and in which context they may become relevant for solving a reverse engineering task.

4.1 Online Platform

Our dedicated online platform provides users with an interface and a set of functionalities, which mimic common interactive disassemblers. The system required users to register an account to allow them to take breaks and perform different tasks at different points in time. After the registration, a "Welcome" page described the various tests and guided the participants through the system's functionalities. The user could then select one of the available tests and proceed with it.

A snapshot of the interactive RE interface is presented in Figure 1. The left panel shows the list of the functions that are present in the binary as well as those imported from external libraries (such as `printf`). When the user visits a function, e.g., `sub_40089a` in Figure 1, the system highlights it with a green bar and displays the list of code cross-references (Xrefs) below the function name. Xrefs are divided into `Xrefs-from` (functions containing a `call` instruction that transfers the control to the visited function), and `Xrefs-to` (functions called within the body of the visited function). The user can control the application by using the mouse (by panning, zooming, clicking on links, accessing a contextual menu with the right mouse button) or the keyboard (by using common shortcuts taken from IDA Pro for moving back and forward, rename variables and functions). The right panel instead is in charge of showing the Control Flow Graph (from now on CFG) of the disassembled function where we decided to resolve library calls with their symbol name and to replace strings' addresses with the string itself. Our UI also includes a call graph view, where the users can visualize the relationship between the function inside the binary.

Enabling the user to have a complete view of all basic blocks at the same time would not allow us to track her progress through the program at the required granularity. Therefore, by taking inspiration from similar experiments performed in the code comprehension community to measure the user attention [8, 26], we decided to implement a *Restricted Focus Viewer* [26] (RFV) solution. This technique provides results comparable with eye-tracking methodologies by dynamically blurring parts of the screen and letting users control the visible area. In our case, only one basic block at a time is readable while all other ones are blurred. For instance, on the right side of Figure 1, the central basic block (from now on BB) is unblurred, whereas the ones above and below are blurred. When the user moves the mouse over a different BB, the system immediately shows its content and re-sets the previous one in a blurred state.

The main disadvantage of this solution is that it can delay the user activity and, as discussed in Section 6.4, it can also prevent rapid glances over different parts of the screen. Even though we are aware of the fact that RFV represents a limitation of our work (as detailed in Section 8), it represents the only solution to precisely measure the basic blocks observed by each participant, while enabling our experiments to be conducted online with users located in different countries.

For each action the user performs over the visible element, the framework generates an event and sends it to our backend server. These events include *New function access* (when the user clicks on a function name), *New basic block visit* (mouse over the new basic block), *Function rename* (right click or keyboard shortcut), *Variable rename* (right click or keyboard shortcut), *Comment* (right click or keyboard shortcut), *Follow the jump to an address* (double click), *Jump to address* (double click on the address), *Move backward* to the previous basic block (keyboard shortcut), *Follow Xrefs-to or Xrefs-from* (click on the desired xref), and *Solution submission* (click on the dedicated link). For each of the cases mentioned above, the web interface generates a JSON request containing a timestamp, the event type (i.e., the action), the position in the binary (i.e., the function address and the BB), and depending on the action type, the arguments. For example, when the humans rename a stack variable, the JSON string will contain the new proposed name and the old stack offset as further arguments.

All the events, along with the user's changes on the code (such as renamed objects, comments, and previously accessed locations), are stored in a database for further analysis.

4.2 Challenges Design

The main problem we encountered when designing our tests was to find a balance between the complexity of the binaries, the amount of data we could collect from them, and consequently the number of people that we could recruit.

Modeling the complexity of a RE task is not easy because a binary could include many features that make the process of understanding its internals more complex. For instance, ob-

fuscated code would require dynamic analysis or access to the binary file for implementing a de-obfuscation algorithm, thus resulting in less data that could be collected by our platform. We also had to design our tasks to be independent from the domain of the different experts; for instance a challenge about packing would be easier for malware experts than for vulnerability researchers. We decided instead to present binaries that implement common functionalities that can be found in any domain.

That being said, there are many potential strategies to provide a measure of the complexity of our tasks. A possible way to accomplish this goal is to rely on the complexity of the source code, as done by [46] to formally describe the difficulty of their binary challenges. Peitek et al., [39] demonstrated, with the use of Functional magnetic resonance imaging (FMRI), the existence of a correlation between such source code metrics and the brain activation registered in users that perform code comprehension tasks. Therefore, we compute a total of twelve metrics (including the Halstead metrics, the cyclomatic complexity, and the number of functions and lines of code) and use these values to assess the difficulty of our assignments. All values for the two assignments are reported in Table 1 (in Appendix). When crafting our challenges we used these metrics of the tasks reported by Sutherland et al. [46] as a lower bound to make sure that our tasks were sufficiently complicated.

After some internal experiments among the authors, we settled for three binaries. Although we understand that three binaries cannot provide a detailed view of the skills that expert reverse engineers acquired after many years of practice, we believe this choice to be a good tradeoff between the amount of data we can collect and the time each participants would need to invest in our exercises.

The first challenge binary was the smallest and only served as a warm-up to make the users comfortable with our tool's interface. Thus, we did not collect data from this first assignment. The other two binaries, which from now on, we will call TEST_1 and TEST_2, were inspired by typical reverse engineering problems in Capture the Flag (CTF) competitions. CTFs are popular games designed to challenge their participants to solve computer security problems. The goal of a RE challenge in a CTF is often to recover the input that needs to be provided to a given binary to produce a specific output. This had the advantage that solutions are small and can be easily verified on our side while still requiring the participants to "understand" the full logic of the target binary. Both the programs were written in C language and compiled for a Linux x64 machine with the `gcc` compiler.

In our tests, all binaries include a *target* function whose purpose is to print the string `'Success!!'`, and the participants were asked to submit a description of the input required by the program to print the success string. To make things more challenging, all binaries were stripped from their symbols and included several "useless" snippets of codes, which had no effect on the problem's solution.

Test 1. The first binary consists of a simple server listening on port 8888 and accepting new incoming connections. For each connection, the server would I) spawn a new process (using the `fork()` function) to serve as a connection handler, II) increment a global counter, and III) invoke the target function that would print the success string if the global counter is equal to three. Therefore three clients need to connect to the server to trigger the `Success!!` string.

Table 1: Complexity metrics of the two assignments

Metric	Test 1	Test 2
Lines of code	146	207
Operators count	426	673
Distinct operators	35	38
Operands count	207	338
Distincst operands	89	87
Program length	633	1011
Program vocabulary	124	125
Volume	4402	7042
Difficulty	39	73
Effort	171678	514095
Cyclomatic complexity	14	19

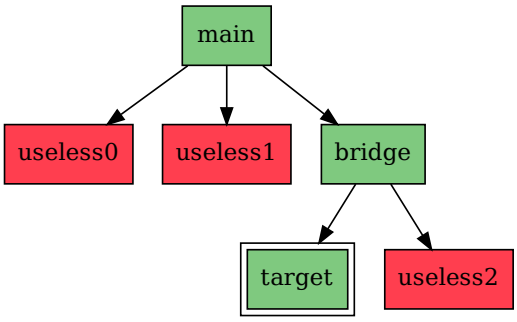

Figure 2: Call Graph of Test 1

The challenge requires the participants to recognize the assembly patterns associated to simple network actions (e.g., the initialization of the socket structures and `bind()`, `listen()`, `accept()` APIs), and the parent/child relationship during a `fork()`. For the sake of clarity, we sketch the call graph of the binary in Figure 2. The figure shows in green the three functions that need to be reversed to solve the exercise, and in red, the three additional functions that play no role in the solution. Two out of the three additional functions are responsible for handling error conditions generated along the binary. The purpose of such procedures is to assess if participants can easily recognize and ignore functions that only generate error messages. The third procedure is the one that implements the connection management.

Test 2. The second binary implements a simple list management application. The application accepts two parameters,

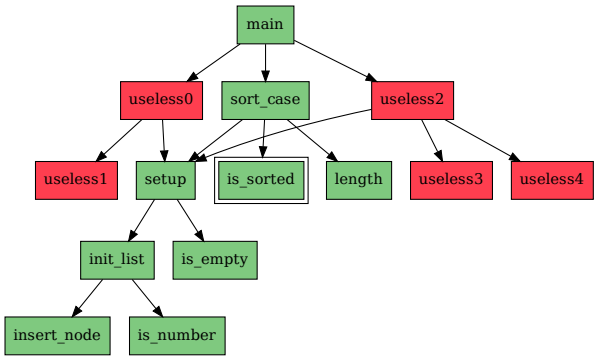

Figure 3: Call Graph of Test 2

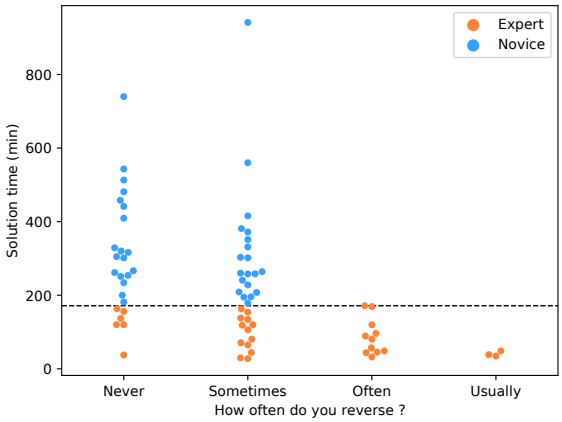

Figure 4: Relationships between how often the subject reverse binaries and the total time spent to solve the exercises.

a list of integer numbers, and one letter that specifies the required operation: (`a`) – the application sums the elements of the list and prints back the result; (`r`) – the application prints the list in a reversed order; (`s`) – the application checks whether the list is sorted and contains at least four elements. If both conditions are satisfied, the program prints the success string.

This second binary is more complicated than the previous one, and all operations are performed over linked lists of custom data structures. To ensure that the difficulty was higher than TEST_1, we verified that all twelve complexity metrics had higher values than in the previous test. The challenge requires the participants to be familiar with linked lists in assemblers (i.e., on the way C `structs` and pointers are compiled in binaries) and to recognize list-related operations (including a bubble-sort implementation). Figure 3 represents the simplified version of the call graph: as in the previous case, we label as *useless* the functions that are not related to the challenge solution. For all the other functions, we report their self-explanatory name. However, the symbols' names were stripped from the binaries, so the participants did not have this information.

5 Participants recruitment

We ensured that all methods and experiments performed for this work are in line with our institutions' research ethics guidelines and our country regulations on data collection and retention. The participants were recruited over a period of several months and the invitation was sent from our institutional email address as proof of credibility. The text, reported in Appendix A, contained a complete description of the experiment with the link to our online infrastructure. As we specify in the recruitment email, we did not provide a compensation for our experiments and we only collected anonymous data.

In particular, we contacted *students* who took a binary analysis or reverse engineering course in three different universities. All students had been previously trained to reverse binary programs, but while some were still beginners, others already had experience by playing CTF competitions. We also contacted nine different top CTF teams, asking for *players* who usually solve complex RE challenges to participate in our experiment. Overall, 95 users responded to our request, but only 72 completed successfully the two tests.

In order to compare the effectiveness of different approaches to read the disassembled code, we split our participants into two groups: *experts* and *novices*. On the one hand, simply relying on the "reputation" of the participants could lead to biased results in our data analysis. On the other hand, self-evaluation questions can also produce biased results because humans tend to adjust their answers depending on their concerns with the interviewer's perception [23, 47]. Therefore, we decided not to divide the participants in two a priori groups, but rather to combine their self-reported experience with the time required to solve the tasks. First, when visiting the website, the participants were asked how often they reverse engineer binary code on a four-point scale in ascending order of frequency: *never*, *sometimes*, *often*, and *usually*. Then, once all experiments had been completed, we identified the time required by the "worst" participant who reported to reverse binaries *often* or *usually* (i.e., 172 minutes). Finally we adopted this value as a threshold: participants who took less time than this threshold are considered experts, otherwise novices. The two groups contained respectively 33 experts and 39 novices. It is worth noting that all CTF players ended up in the expert group.

Figure 4 shows the relationship between the answer to the frequency question and the time required to complete the two assignments for the two classes of users whereas the dashed horizontal line represents the threshold we have inferred.

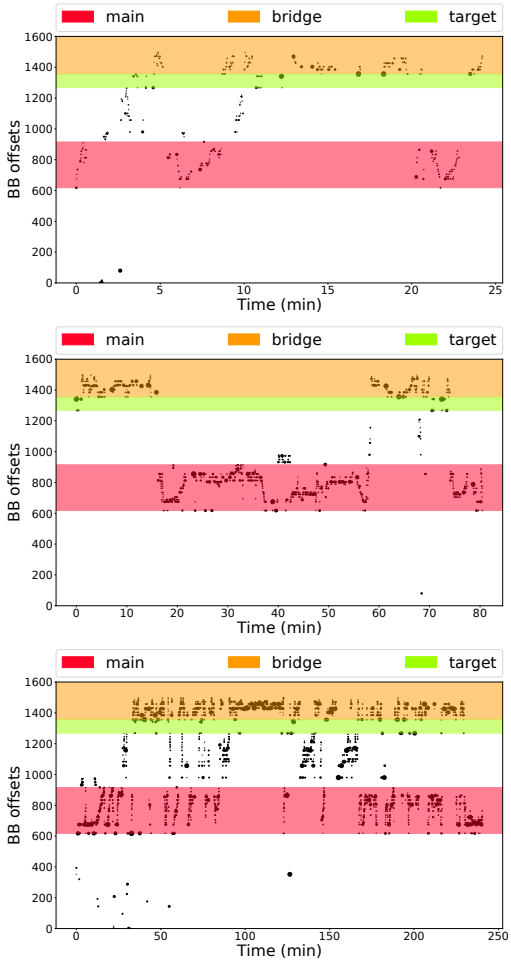

Figure 5: Three distinct RE sessions of Test 1 showing the time spent on each basic block during the session

6 Data Analysis

We now discuss the results of the participants who completed the two exercises (39 novices and 33 experts). First of all, as we expected, novices and experts spent a different amount of time to complete the assignments. In fact, the two exercises combined took between 24 and 172 minutes (92 on average) for the subjects in the experts' group and between 178 and 941 minutes (340 on average) for novices. In other words, even though the exercises were relatively simple, beginners were, on average, 3.7 times slower than experts, and the fastest beginner was 7.4 times slower than the fastest expert.

To avoid bias, we computed the confidence intervals for the two groups of users, with a confidence value equal to 0.95. In this second scenario we obtained that the time required was between 75 and 110 minutes for experts whereas it ranged from 289 to 391 for novices.

Moreover the application of a 2-sample t-test over the two groups in relation with the solution time confirmed us that it is meaningful to separate the two groups in terms of time needed to accomplish the task (t-test 9.31, p-value 3.5e-10).

We also analyzed the solution time by splitting the users according to their answer to the initial question about how often they reverse binaries.

As shown in Figure 4, it took on average 301 minutes for users who answered 1 (rarely reverse binaries), 233 for those who answered 2, 86 minutes for those who answered 3, and finally 40 minutes for the only three experts who reported to reverse binaries on a daily basis. This shows that while all participants in our expert group were fast, on average, those who perform this task more often tend to be faster.

Mining for Strategies

We started our analysis by manually inspecting the telemetry data collected from the users' sessions, looking for macro-differences that could indicate the use of different strategies. As an example, Figures 5 shows a graphic representation of the behavior of three users during the first exercise (time is on the X axis and BB addresses on the Y). The horizontal bands of different colors represent the three useful functions (those required to solve the exercise), while the white region indicates the BBs located in other irrelevant parts of the program. Each dot corresponds to the user focusing on a given basic block for a certain amount of time (expressed by the size of the circle). The labels `target` and `bridge` respectively indicate the function that prints the success string and the function that has `main` as the caller and `target` as one of the callees.

The first two graphs belong to experts (the fastest in our test and an average one), while the bottom depicts a beginner session.

The three graphs clearly show very different approaches to reverse the same binary. The second expert spent a considerable amount of time on `main` (the red band), while the first moved away from it after a few minutes and returned to its code only after a first overview of the binary. Moreover, the order of their visits is different. The first started from `main` while the second user started the exploration from the `target` function (where the success string is printed). However, even if the first approach is more efficient, the first expert spent more time looking at unrelated code (dots in the white band) than the second (9 against 4 minutes).

The novice session appears more chaotic. It contains many more points (i.e., BB visits), reaching a total of 3469 visited blocks, and the user kept switching back and forth between the three main functions, probably trying to make sense of the entire program.

Looking at all 72 graphs, it seems like everyone has their own style. However, we are interested in generalizing these first observations and finding whether the strategies adopted by experts have something in common that does not appear in the novice sessions. We also notice considerable variance among the experts themselves, so we want to study possible

Table 2: Prevalence of Function-level Strategies for novices and experts while approaching the two binaries

Strategy	Test 1		Test 2	
	Novices	Experts	Novices	Experts
Sequential	4	-	8	-
Backward	2	6	5	8
Forward	33	27	26	25
Depth-First	0	2	1	6
Breadth-First	11	8	16	12
Hybrid	28	23	22	15

differences among users in the same group.

To perform this analysis, we first distilled the collected low-level events into several high-level features representing observable behaviors that we could identify in our dataset of participants. We then tested whether each feature was substantially different between experts and novices and whether it was positively correlated to the overall solution time. While this second aspect does not necessarily imply causation, it can still show which set of techniques are more commonly used by those reversers who could complete the exercise in a shorter amount of time. Before performing the statistical test, we checked if the data distribution (especially for time samples) was normal and, in negative case, we applied a log-transformation to normalize it. For each test that we executed, we collected the resulting p-values inside a vector, and we used the Bonferroni method to correct them with an input alfa of 0.05 (all additional hypotheses we tested are listed in Appendix 7). The corrected alfa that we obtained is 1.2e-03 and all values that we report in the paper already take into account the Bonferroni correction.

6.1 Functions Exploration

Function exploration strategies play an important role to discover the path between the `main` and the `target` functions. Once this path has been unveiled, users can focus on the BBs that compose the functions in this path and therefore they abandon their function-level strategy and drive the exploration according to what they found. For example, if we consider the second expert of Figure 5, we can note that she adopts a backward approach, starting from the `target` and then reaching the `main` function. Then, she focuses with more attention on the BBs of such (and other) functions to figure out how to craft the proper input to solve the challenge.

Three different ways exist to move across functions: by following Xref, by direct access (i.e., by clicking on the function name in the sidebar), and by following the CFG (i.e., by clicking on the `call` instructions or by using the `ESC` key to step backward). Accordingly, we identified three main exploration strategies: *forward* (starting from the program's `main` and following the CFG), *backward* (by first searching the API call that prints the `SUCCESS` string and then backtracking the analysis by following `Xref` references), and *sequential* (i.e., by exploring each function independently of its role or position in the callgraph).

Whenever a user explores the code of a function and encounters a `call` instruction, she can decide to proceed either *depth-first* or *breadth-first*. In the first case, the reverser visits each function vertically until she reaches a leaf. In the other, she explores the called functions horizontally before moving deep into each part of the call graph. To discern between the two strategies we cannot use standard DF and BF detection algorithms, as users often alternate between the two methods. Therefore, we considered a sliding window of two visits on the call graph and compared consecutive bi-grams by looking for typical BF or DF patterns. For instance, a typical window of a user using a DF approach consists of two visits to different functions following the direction of the graph's edges. On the other hand, the bigrams of a BF strategy contain consecutive bigrams of the same two functions, but appearing in alternate directions (e.g., $f-g$ followed by $g-f$).

We say that a participant predominantly uses a given strategy if it employs it at least 50% more frequently than the other. When this does not happen, we assign the user to a *hybrid* category, which means that the reverser adopted both exploration strategies at different points in time without a clear preference. The prevalence of the different exploration techniques is summarized in Table 2 for both novices and experts, and it shows that experts and novices clearly use different techniques. The sequential exploration is adopted by a non-negligible amount of the beginners (4 in the first test and 8 in the second one), but none of the more experienced reversers follow this approach. Users in both categories prefer the forward rather than the backward exploration. We can also see that BF visits are much more common than DF, and it is important to note that almost none of the novices resorted to a DF approach.

So far, we learned that experts tend to use different strategies, but it is still unclear whether a given strategy impacts the time required to solve the exercises. We performed an ANOVA test by splitting the participants' solution time into 3 groups (*depth-first*,*breadth-first* or *hybrid*), and applying the `one way` function to these. We ran a separated test for each challenge because some participants changed their strategy depending on the task, but all tests failed (p-values for each challenge were 0.17, 0.19 with effect sizes of 1.8 and 1.6 for the *forward-backward-sequential* classification and 0.14, 0.2, effect sizes of 2.1 and 1.9 according to the *depth-breadth* separation). In fact, as depicted in Figure 6, all techniques were used to efficiently solve the two exercises.

6.2 Code Selection

We now check *where* the reversers spent most of their analysis time. Table 3 shows all functions in the second binary, and for each of them, it reports several metrics. The table is divided into two parts: the top half lists useful functions, i.e., those involved in the solution of the problem. The bottom half

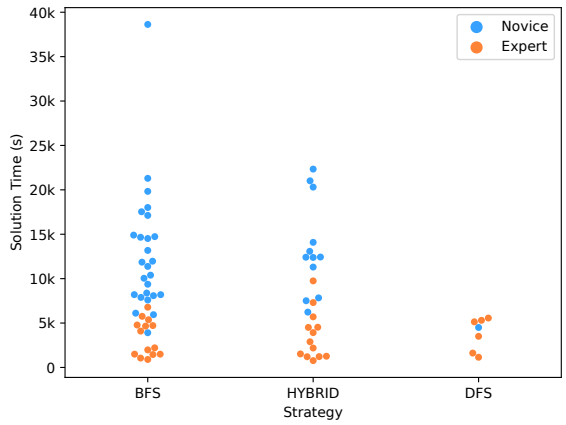

Figure 6: Time needed to solve Test 2 grouped by strategies.

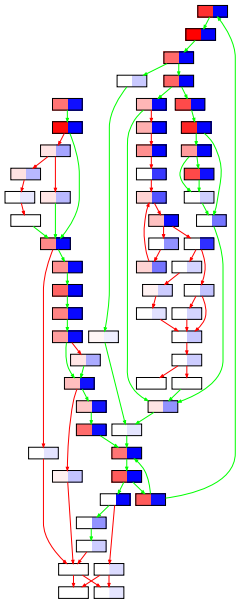

Figure 7: Average times spent in the BBs of Test 1

lists instead the five 'useless' functions (the binary accepts three different commands, but only one is required to print the success string). However, since also the related functions include irrelevant paths (e.g., to handle error conditions), in the first two columns we report the total number of basic blocks in the function and the total number of 'good' blocks (B_{good}), which are those that must be reversed to conclude the exercise. The table also reports how much time (both the absolute median time and in percentage over their entire session) experts and novices spent on each function and the overall ratio between the experts and the novice time (last column) computed as the absolute median time of novices divided by the absolute median time of the experts for that function.

Table 3: Median Time Per Functions for task 2

Function	BB	BB_{good}	Experts min (%)	Novices min (%)	Time Ratio
main	16	12	9.1 (15.8%)	29.4 (13.9%)	x3.2
sort_case	8	6	5.7 (9.6%)	18.2 (8.6%)	x3.1
setup	6	4	4.4 (7.8%)	13.4 (6.3%)	x3.0
is_sorted	12	10	10.4 (18.1%)	28.9 (13.7%)	x2.7
init_list	8	7	8.7 (15.1%)	38.7 (18.3%)	x4.4
is_empty	1	1	0.26 (0.4%)	1.0 (0.5%)	x3.9
insert_node	7	6	5.8 (10.2%)	28.0 (13.4%)	x4.8
is_number	9	8	4.7 (8.2%)	22.9 (10.9%)	x4.8
length	4	4	3.5 (6.2%)	12.0 (6.0%)	x3.5
useless-0	6	0	1.0 (1.8%)	2.8 (1.3%)	x2.8
useless-1	4	0	0.5 (0.9%)	2.9 (1.4%)	x5.7
useless-2	7	0	0.9 (1.6%)	2.9 (1.4%)	x3.1
useless-3	4	0	1.5 (2.6%)	5.4 (2.6%)	x3.6
useless-4	4	0	0.8 (1.4%)	3.9 (1.9%)	x4.7
TOTAL	96	58	57.2 (100%)	210.4 (100%)	x3.6

There are two interesting observations we can make from these results. First of all, all participants spent most of their time on *main* (because it was longer) and on the functions that operate on linked lists. However, beginners were impacted more by the nature and complexity of the function. For instance, they spent much more time (4.8x slower than experts) to recognize that `is_number` only verifies that all parameters are integer numbers. We believe that this is due to the fact that similar simple functionalities are encountered frequently by reversers and therefore are easily recognized by experts.

Nevertheless, the most striking result is the fact that, in percentage, novices spent almost the **same** percentage of their time (8.6% vs 8.3% for experts) on reversing useless code (even if in absolute terms they still spent four times more than experts). At first, this seemed counter-intuitive. In fact, we expected experts to be better at quickly skimming through the code and ignoring it if it was not related to their task. However, given the numbers in Table 3, we hypothesized that this discrepancy is because novices were so slow to understand the difficult parts of the code that, in percentage, they appeared faster in discarding the non relevant ones. We computed the same values for the first binary and we observed the same trends even if in that case the number of functions is minor compared to the second challenge (only 6). Indeed, the *main* is still the function where users spent most of their time and the effort dedicated to the useless functions is basically the same in percentage (13.1% for experts vs. 12.5% for novices). For space reasons we report the values in Table 6 in the Appendix.

Hence, we decided to measure the total number of basic blocks that were visited by each participant. In total, the two exercises combined contained 155 basic blocks, but only 94 (61%) of them were actually along the solution path. To complete the two exercises, the median expert completely skipped (i.e., never even checked once) 24 basic blocks, while the median novice skips only 6 of them. Indeed, this fact shows that experts could cut entire branches (or functions) by

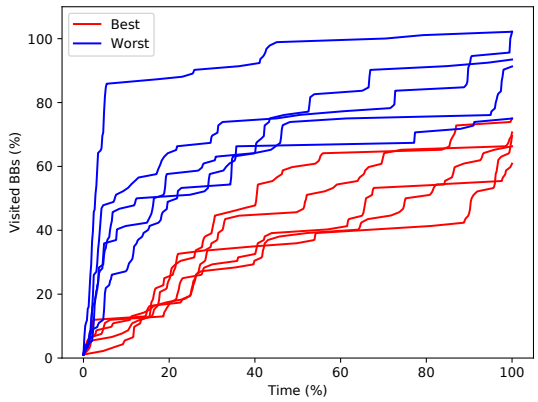

Figure 8: Progression of Top5 and Bottom5 experts in the second challenge.

only looking at a few of their blocks.

For instance, Fig. 7 shows the CFG of Test 1. The green edges point to interesting BBs while the red ones point to useless BBs. Each node is split in half: the intensity of the left side represents the amount of time spent on that BB by the experts (on average); the right side represents the same for novices. If we consider the noninteresting paths, the blue intensity is generally higher than the red. Experts mostly recognize that some code parts lead to useless BBs by just reading the first BBs of that function and then recovering the correct path to the target function. Novices instead needed to go through also the noninteresting parts of code before understanding that they do not need them for their purposes.

Finally, we performed a 2-sample t-test using as an hypothesis the correlation between the group (i.e., expert/novice) and the time spent on non-useful portions of code. With a p-value of 5.3e-04 and a t-test of 4.86 we can conclude that indeed there are statistically significant differences in the way the two groups of participants look at the non-interesting parts of the binary.

6.3 Birdseye Overview

Experiments on code comprehension conducted by Uwano et al. [51], and independently validated by [44], have found that users often perform an initial scan of the entire codebase to get a general idea of what the program is supposed to do. During this initial scan, the authors found that programmers went through 70% of the code in the first 30% of their analysis.

By looking at the reverse engineering sessions we collected in our experiments, we can clearly identify some reversers performing such preliminary scans. However, this behavior is not as typical as one might expect. In fact, in our data, only 36.0% of the experts visited 70% of the code blocks in the first 30% of their time. On average, at the 30% mark, expert reversers had visited only 48.2% of all BBs. The number increases to 53.4% (still well below the 70% threshold) if we only count the good basic blocks and ignore those that were not relevant for the task. Beginners tended instead to move through each BB much quicker at first and to return back multiple times during their sessions to read again the code (we will analyze this aspect in Section 6.4). As a result, 69.4% of them met the 70% threshold at the 30% mark.

But there is more. Figure 8 shows the Cumulative Distribution Function (CDF) of the visited BB over time by comparing the top five experts (based on their solution time) against the bottom five. It is interesting to observe that the fastest reversers (in red) progressed more linearly and did not employ any initial survey strategy.[2]

This seems to suggest that a preliminary overview of the entire binary might be useful to get an orientation in large codebases, but it might not be very useful in smaller exercises. Even more surprising, we found that the majority of experts did not even 'try' to quickly skim through the code of the various functions, even though they did not know in advance anything about the complexity of the task.

Figure 8 also confirms what we found in the previous section, i.e., that all best reversers were not fastest only because they could read and understand the code faster, but also because they reversed **less** code. On the far right of the CDFs we can see that the red curves terminate between 60% and 80% of the total BBs (remember that only 61% were along the solution path), while the blue lines fall in the range between 80% and 100%.

6.4 Basic Blocks Exploration

After looking at the function granularity, we now focus our attention on individual basic blocks.

Thanks to the use of the *restricted focus viewer*, our reverse engineering platform can accurately track the time spent by each participant on each individual BB. However, not all these time events are equally important. For instance, it can occur that when moving the mouse pointer between two BBs, the user accidentally moves the mouse over an intermediate BB without being really interested in its content. Our infrastructure would capture this behavior, generating an event for all three BBs. To remove the noise introduced by these spurious events, we decided to conservatively discard all the views with a duration below 500 milliseconds. This threshold is based on the fact that, according to Rayner et al. [42], while reading text, the eyes stay upon each single location from 100 ms to over 500 ms. Given the fact that a BB is often composed by multiple lines, this threshold ensures that a participant had time to focus on at least one location in the BB. Anything below that would not provide much information to the reverser.

It is essential to understand that the time a reverser spends on a single BB is affected by multiple factors, including the

[2]This trend does not change if, instead of basic blocks, we perform the measurement at a function granularity (we omit the graph for space reason).

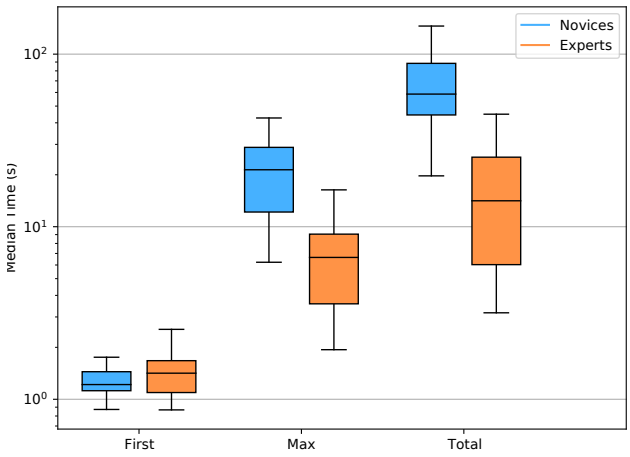

Figure 9: Comparison of the distribution of T_{First}, T_{Max}, and T_{Total} time among the users in the two groups.

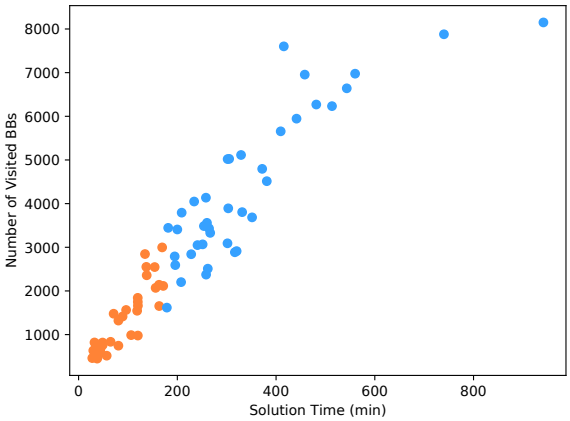

Figure 10: Solution time w.r.t. number of visited BBs.

BB complexity, the user assembly reading skills, the role of the block inside the binary, the navigation strategy of the user, and the state of the ongoing RE session. We will try to break down these factors in the rest of the section.

To begin with, for each BB we identify three different time values. First, the time each user spent on the block the first time she encountered it (T_{first}). Second, the total cumulative time (T_{tot}) each user spent on the BB over the entire exercise. And finally the longest consecutive time each user spent on the block (T_{max}).

By comparing these three time intervals, we can make several interesting observations. Figure 9 shows the distribution of the median time spent by each user over all the basic blocks of the two exercises. It is interesting to note how, the first time they encounter a new basic block, both experts and novices spend only a few seconds on its code: on average 1.3s for beginners and 1.5s for experts. Instead, the maximum and total time spent on the blocks are over one order of magnitude higher, often lasting for tens of seconds (6.8s vs 21.9s for T_{max}, and 16.3s vs 73.4s if we compute the median times for the T_{tot}). As a confirmation of this aspect, we ran the 2-sample t-test over the values of T_{first}, T_{max} and T_{tot} collected over each user and then separated by novices and experts. Indeed, we obtained that the difference for the first visit (T_{first}) is not statistically significant (p=0.2) but the time difference on T_{tot} and T_{max} are (respectively with p=7.4e-07 and p=1.5e-08).

At first, one might easily dismiss the role of these first short visits, nothing more than a quick glance at a block while the user rapidly moved the mouse over it. It might seem obvious that the 'real' reverse engineering is performed over the subsequent visits. However, if we compute the fraction of BB that a user visited only once we see that things are more complex. On average, experts visit 28% of the BBs only once. In 80% of these cases, the visit lasted less than two

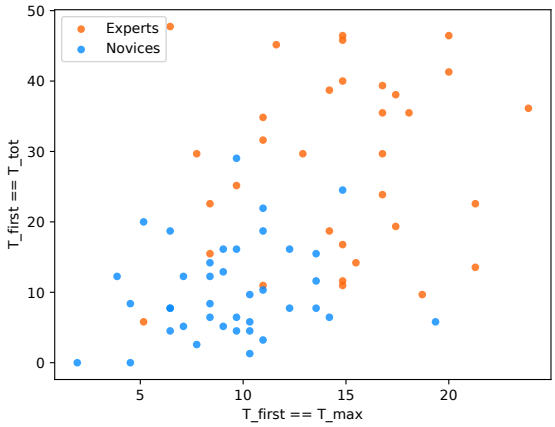

Figure 11: Percentage of Basic Blocks visited only once, or analyzed on the first visit.

seconds. This means that experts dismiss almost 22% of the basic blocks in a single glance. On the contrary, inexperienced users make a single visit only for 10% of the BB, and in total, dismiss only 7% in less than two seconds.

All the remaining BBs are visited by each reverser multiple times. In fact, even if the two programs combined contained only 155 BBs, to complete the two exercises, experts visited, on average, 1368 basic blocks and novices 4326 (2-sample t-test=9.7 and p=6.8e-12). Figure 10 shows the relationship between the time required to solve the challenges and the number of visited blocks.

However, visiting a block multiple times is not always a sign of inefficiency, and in some instances it is even unavoidable (e.g., those blocks that contain a function call are often re-visited when the user moves out of the function and back to the callee). We ran the Pearson correlation to test if a

Table 4: Correlations between visits duration and BB length

	Experts		Novices	
Hypothesis	**Pearson**	**p-value**	**Pearson**	**p-value**
T_{max} and $len(BB)$	0.29	1.0e-04	0.31	5.8e-04
T_{tot} and $len(BB)$	0.30	8.28e-04	0.33	1.6e-04
T_{first} and $len(BB)$	0.37	1.9e-05	0.37	1.3e-06

relationship exists between the number of times the users go through an already visited BB and the overall solution time and obtained a result of 0.68 (p-value 1.2e-05) for experts and 0.46 (p-value 2.5e-04) for novices. Therefore we investigated this aspect in more detail and computed the number of times the first visit to a basic block was not the only one, but it was the longest (i.e., $T_{first} == T_{max}$).

If we assume the most prolonged visit is when the user actually completely reversed the BB code, we can use this indicator to know whether this is performed for the first time the reverser encounters a new code. In this case, the median is 9.6% of the BB for beginners and 14.8% for experts. Again, it seems that experts tend to fully understand the code the first time they read it, while beginners go back multiple times, and in 80.6% of the cases, their first visit is not the one where they reason the longer on the code.

Finally, it is interesting to test if these short first visits are just a consequence of the fact that a reverser might be simply faster at processing assembly code. In other words, we wanted to test whether those users that have shorter first-time visits (T_{first}) also spend less time overall on the BBs (T_{tot}). However, the Pearson correlation of the 2-time values is -1.2, p-value=0.5, showing no statistically significant correlation.

Figure 11 shows how a scatter plot of the two aforementioned metrics (percentage of blocks visited only once, and for which the first visit is the longest) can clearly separate the majority of the experts from novices reversers.

6.5 Speed Factors

In our final analysis of the different reversers' speed, we look at which factors affected the time spent on individual basic blocks.

For this purpose, we limit our analysis to those blocks that actually needed to be understood in the first place. Thus, we first remove those BB that are NOT related to the solution of the exercise as well as all headers and footers of the functions (as it might not always be required to analyze their behavior carefully). The remaining (which we will refer to as BB_{core}, and that account for 47% of all blocks in the two assignments) capture the code each user *had to reverse* to reach the correct solution.

The first hypothesis that we wanted to formulate was to study the potential correlation between the time spent on each block and the size of the block itself. Indeed, we observed that the first, total and max times are positively correlated to the number of assembly instructions contained in the basic block. However, the exciting result is that the Pearson correlations are quite small for T_{tot} and T_{max} while they exhibit an higher value for T_{first} as reported in Table 4. Moreover, under the same hypothesis, the correlations are always more elevated for novices.

Table 5: Statistical tests w.r.t. the branch selection data (upper part) and the semantical elements data (lower part)

Hyphothesis	**Result**	**p-value**
Novices true branch & solution time	0.21	0.06
Novices close branch & solution time	0.13	0.2
Experts true branch & solution time	0.42	0.7
Experts close branch & solution time	0.87	0.4
2-sample T-test Comments	0.4	0.6
2-sample T-test Variable Renames	0.8	0.4
2-sample T-test Function Renames	0.7	0.4

One way to interpret these results is that the amount of time spent by reversers on a basic block is only marginally influenced by the time required to actually read (or *'parse'*) each assembly instruction. The impact is more visible for inexperienced reversers (who probably spend more time reading the assembly) and less on experienced users. To understand which other factors contributed to the reversing time, we extracted the top 5% of the basic blocks in which each user spent most of her time. Then we compared all sets to identify those blocks that were problematic for a large percentage of users.

If we look at total or max time, both experts and beginners spent most of their time (respectively 19% and 18% on average) on blocks that prepared the function call parameters. While usually straightforward to reverse, all reversers probably paused to reason about which values were passed to the function's parameters. If we look instead at the blocks that frequently appear among beginners but not among experts, we find a total of 20 BBs that are shared between a minimum of 3 and a maximum of 11 novices and that are responsible for an additional 9% of time on average overall. We analyzed them to unveil the assembly language (ASM) patterns that slowed down the novices while reading them. In total 6 blocks contain uncommon instructions (such as `setnz`, `imul`, and `sar`) and 7 include instructions that operate with in-memory data structures, thus requiring to reason about the memory layout of the program in that specific moment (e.g., instructions that access the i-th element of the list of Task 2). We also found 3 BBs that operate on the static strings contained in the binary. Among these 20 BBs only 4 of them have a number of instructions major than 10 while the other 16 contains less than 6 instructions (and in 10 cases they were just 3 instructions long). This finally shows that the nature of the instructions is more relevant than their number to explain the comprehension time for beginners.

6.6 Other Aspects

In the previous sections, we discuss several aspects we believe can capture subtle but essential characteristics of the behavior of either experienced users or beginners. We also tested many other hypotheses and tried to isolate other behaviors (reported in Appendix 7) but for which we could not find any statistical difference among our users. For these hypotheses that did not find a statistical validation we report the p-value that we obtained after running the Pearson correlation, but we omitted the correlation value itself for space reasons, since it was not meaningful. However, we want to add two more short points to our analysis regarding the impact of the user interface in branch selection and the other events we collected from our platform.

Branch Selection - when visiting a conditional BB for the first time, beginners choose to explore the true branch first in the 41% of the cases, whereas experts followed the true branch in the 42%. However, we found that the physical position (on screen) of the basic block is much more important than its logical one. In fact, our results show that both experts and novices tend to simply visit first the closer basic block, respectively in 87% and 88% of the cases they encounter a branch. Finally we tested the hypothesis that the choice of either true branch or close branch as a next step has an effect on the overall time to reverse engineer the binary. However for both experts and novices we obtained p-value > 0.1 (values are reported in Table 5).

Comments and Rename Actions - we also investigated the use of the other features implemented in our infrastructure: comments, variable renames, and function renames. On average, we recorded 24 comments among all the expert sessions, whereas we count only 11 from novices. The same trend happens for variable renames (19 vs. 7) and function renames (12 vs. 2). One more time we applied the 2-sample t-test for each of the semantical elements created by the user, divided for experts and novices. The results of the test (reported in Table 5) show no statistical significant relationships between these features and the users performance. At a first look, this result looks like surprising as we would expect that a statistical significancy exists between the usage of semantical elements, the solution time and the experience level. However our hypothesis for this behavior is that probably the statistical relationship between the use of semantical text fragments and the RE performances become more and more evident while observing this on larger and more complicated codebases (potentially together with other reversers with the same experience and working in the same team). We will discuss more carefully about challenge design limitations and future directions respectively in Sections 8 and 9.

7 Summary of Findings

In this study we quantitatively measured the behavior of 72 reversers, both experts and novices, over a total of 272 hours of RE activity. By looking past the individual features discussed in the previous section, we will now summarize the main findings that emerged from all our results.

First of all, we found that each user is unique and has her strategy and her way to reverse binary code. However, by looking under the apparent diversity of actions, we can identify a number of core strategies. To begin with, novices move prevalently forward from the program's `main` while experts mix forward and backward movements. While statistically the difference is clear, there are notable exceptions in all groups, showing that one can be very efficient independently from the strategy it adopts (except for the sequential scan that is only used by the very beginners).

Experts also exhibit a more linear progress, avoiding to jump back and forth among the same basic blocks they already visited in the past. Moreover, they make every visit count, even the first one. This allow them to dismiss 22% of the basic blocks in a single observation, which often last less than two seconds. The 70-30 birdseye scan observed several times in studies of program comprehension does not seem to apply to binary reversing, at least at the small scale dictated by our exercises. Instead, the experts' ability to quickly identify and *ignore* the regions that were not relevant for their task was one of the essential aspects that distinguished experienced users from beginners. This, which fits the self-reported techniques that Votipka et al. [53] group under the name of *subcomponent scanning* could, in fact, be related to the ability of the expert's brain to recognize code patterns, but more focused experiments (e.g., with brain EEG sensors) are needed to investigate further and validate this hypothesis.

Finally, our experiments show that the number of instructions is a very poor predictor of the time required to understand a piece of code and that the presence of less common instructions has a more noticeable impact only on novices.

8 Limitations

When we designed our experiments, we had to make many choices to balance the difficulty of the problems (and, therefore, the time required for completing the exercises), the amount of data we could collect, and the impact of our instrumentation on the user experience. These choices might have introduced biases in the results or might have prevented us from observing some aspects of the users behavior.

Expertise - In our study, we measure the expertise of a user in three ways. First, based on "reputation", i.e., by inviting as experts only those users that can already solve very difficult reverse engineering challenges. Second, by the frequency on which each user reverses binary files (as reported in the questionnaire during registration), and finally by the total time required to solve the two assignments. However, one may argue that a good reverser does not necessarily need to be fast—but some may prefer instead to be meticulous and

precise in her findings. New experiments should be designed only for expert reversers to measure this aspect by providing them with more challenging assignments where precision may be more important than speed.

Restricted Focus Viewer - The use of a RFV to capture the part of the code a user is currently focusing on is a standard methodology in comprehension experiments. While it allows for remote participation without the burden of on-site (and uncomfortable) eye-tracking solutions, this choice also introduces some limitations. First, it impacts each participant's overall speed. It also prevents glances, in which users quickly look at a different basic block, maybe just to check a register or the final instruction. In our settings, this requires moving the mouse, and therefore users might perform this task less often than in an unconstrained environment. We can also hypothesize that the issue with the glances affects the order in which basic blocks are visited. Another potential drawback is that it could technically discourage the participants from using the birdseye overview (Section 6.3), forcing them to rely mostly on their own memory to remember a previously visited basic block. However, this affects only a reduced number of cases: moving the mouse back to a previous basic block is "expensive" only if we want to quickly recall a specific location of that block (e.g., a register, a single ASM line) as in the case of glances, but it becomes fundamental, therefore justifying the time "expense," if the participant wants to entirely read the BB.

These factors can affect the code comprehension process by distorting the way it is performed. In absence of RFV, we could expect a higher number of glances and therefore a shorter time to discard some blocks of code. Unfortunately the only way to determine how the RFV influences our findings would be to compare it with some data collected using the same tool without RFV, which is impossible by design. Thus we can only acknowledge this limitation and hope that future studies will be able to overcome it with different technologies or with a different experiments' design (e.g., smaller group of experts monitored with eye tracking devices).

Nature/Number of the Exercises - It is possible that the tasks we ask the participants to perform may affect the ecological validity of the behaviors we observed in their session. In particular, more difficult problems and larger codebases could require different strategies or help identify other aspects that differentiate one expert from the others. However, in this measurement study, we wanted to include beginners and, therefore, opted for tasks that could be solved (even if with more significant effort) by non experienced reversers. While the number of tasks could be extended , this would increase the time to complete our assignment, especially for some participants who already spent several hours with the current configuration (and that are not affiliated to our group). Even if this represents a limitation of our work, it is probably an inevitable choice given our initial goals, i.e., to involve many users ranging from the "newbie" to the "elite" hacker and compare them on the same set of challenges. We hope that future studies will either confirm (or disprove) our results with larger and more difficult binaries to reverse. For example, we can hypothesize a more frequent use of the birdseye overview (described in Section 6.3), which in our experiments was used only by a small percentage of experts. Another aspect that is largely related to the size of the binaries is the number of functions, and therefore we expect a more pronounced impact of the different strategies described in Section 6.1 on larger programs. For instance, an initial horizontal investigation can be beneficial when analyzing larger codebases.

9 Implications and Future Work

The first area that would benefit from a solid comprehension of the RE process is teaching and training. Several features we identified are correlated with experience, but this does not mean that could not be improved by performing specific exercises. As of now, RE learning is mainly based on the resolution of binary challenges of increasing complexity [14]. A possible implication of our findings could be to design binary analysis exercises more focused on a few BBs, to stimulate a student to match and memorize patterns. Concerning these ideas, currently, we are trying to integrate them into the RE courses organized by our institute.

If teaching RE to humans is essential to form new experts in the RE domain, training computers to mimic human behavior would be fundamental to scale over the large amount of software/malware released every day. We believe that studying the techniques used by humans is the first step to discover new ways to train machine learning models to perform similar tasks. Psychologists have learned that many activities are inherently linked to the ability to recognize previously seen patterns [19,31,55] and that the experts are those who learned a significant number of patterns over several years of experience. Since learning to recognize patterns is what ML algorithms can do well, we also studied which aspects human experts focus their attention on to provide the building block for further studies on such topics. Extending the concept, we could even introduce semantic awareness in the classifier. For instance, many experts in our experiments could easily recognize non-standard implementations of the list operations related to the second challenge or discard branches/functions by just reading a subset of the related BBs. This suggests that ML classifiers could be trained to mimic this behavior and to automate the pattern recognition phase both for useful and useless portions of code.

Moreover, even though our study does not put a particular emphasis on the usability issues, we deem that some insights (such as the fact that proximity in the UI view guides the reversers' exploration) can also improve the current interface of reverse engineering tools to make the human activity even more effective.

Finally, we can hypothesize implications of our work with research fields that do not necessarily fall under the RE category. For instance, authors of [13, 52] conducted a user study

about the effort needed to violate source code obfuscation techniques. In this context, one of their limitations was that they could not perform fine-grained measurements. Therefore, we argue that our methodology (i.e., the use of RFV with its pros and cons) could provide meaningful input in these scenarios.

The presented ideas and implications are only a subset of the possible consequences of more advances in the RE field. Indeed, as explained in the limitations (Section 8), our measurements came at the expense of many restrictions in the scope of the study. Future work will have the role to focus on the many aspects that remained uninvestigated, thus offering a broad range of research directions that can result in even more implications. The first thread is to dedicate a set of experiments for a group of expert-only participants considering more complicated challenges. Besides that, another branch is definitely to focus on other RE aspects which belong to a more specific domain, such as malware analysis or vulnerability discovery. Also, the methodology will play a fundamental role in future research, preferring remotely accessible solutions for studies over a large group or eye-tracking devices for smaller groups. We hope, with our work, to provide significant input for many other papers about this topic that still has so many questions that need to be answered.

Furthermore, our study presents itself with an exploratory spin, which justifies why we tested many hypotheses along our road. It is important to understand that such hypotheses (also those ones reporting p-value = 0) represent for now promising theories that need more validation before becoming actual findings. Therefore, because this field is still in its infancy, we invite future researchers of this topic not to assume our isolated behaviors as the final word, but rather to validate/invalidate them with different experiments and methodologies.

10 Conclusions

Drawing inspiration from the first set of interviews conducted by Votipka in 2020 [53], the objective of our study was to lay the second brick towards a solid understanding of the RE process from an assembly code comprehension perspective.

A deep understanding of the topic can help us from different points of view and has a few interesting implications that should be taken into account. With our work, we hope to provide a valuable input for future research in a field that, so far, was poorly explored.

In the spirit of open science, we release [3] the source code of our web RE framework together with the challenges and the test scripts, to allow the community to continue further studies in this direction. Lastly, our measurements are summarized in Table 8 in the Appendix.

[3] https://github.com/elManto/REmind

Acknowledgements

We would like to thank the anonymous reviewers for their constructive feedback, and Slasti Mormanti for sharing his experience with us.

This research was partially supported by the European Research Council (ERC) under the Horizon 2020 research and innovation program (grant agreement No 771844 BitCrumbs) and by the Defense Advanced Research Projects Agency (DARPA) under grant agreement FA875019C0003.

References

[1] Defcon ctf final scores. https://www.defcon.org/html/defcon-24/dc-24-ctf.html, Accessed September 28, 2021.

[2] Ghidra. https://ghidra-sre.org/, Accessed September 28, 2021.

[3] Ida pro. https://www.hex-rays.com/products/ida/, Accessed September 28, 2021.

[4] Andronicus A Akinyelu and Aderemi O Adewumi. Classification of phishing email using random forest machine learning technique. *Journal of Applied Mathematics*, 2014, 2014.

[5] A Anneliese von Mayrhauser and Steve Lang. Program comprehension and enhancement of software. In *In Proceedings IFIP World Computing Congress-Information Technology and Knowledge Engineering*, 1998.

[6] Vairam Arunachalam and William Sasso. Cognitive processes in program comprehension: An empirical analysis in the context of software reengineering. *Journal of Systems and Software*, 1996.

[7] Liam J Bannon. From human factors to human actors: The role of psychology and human-computer interaction studies in system design. In *Readings in human–computer interaction*. Elsevier.

[8] Roman Bednarik and Markku Tukiainen. Visual attention tracking during program debugging. In *Proceedings of the third Nordic conference on Human-computer interaction*, 2004.

[9] Roman Bednarik and Markku Tukiainen. An eye-tracking methodology for characterizing program comprehension processes. In *Proceedings of the 2006 symposium on Eye tracking research & applications*, 2006.

[10] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International journal of man-machine studies*, 1983.

[11] Adam R Bryant. *Understanding how reverse engineers make sense of programs from assembly language representations*. PhD thesis, Air Force Institute Of Technology, 2012.

[12] Jill Cao, Scott D Fleming, Margaret Burnett, and Christopher Scaffidi. Idea garden: Situated support for problem solving by end-user programmers. *Interacting with Computers*, 2015.

[13] Mariano Ceccato, Paolo Tonella, Cataldo Basile, Paolo Falcarin, Marco Torchiano, Bart Coppens, and Bjorn De Sutter. Understanding the behaviour of hackers while performing attack tasks in a professional setting and in a public challenge. *Empirical Software Engineering*, 2019.

[14] Tom Chothia and Chris Novakovic. An offline capture the flag-style virtual machine and an assessment of its value for cybersecurity education. In {*USENIX*} *(3GSE 15)*, 2015.

[15] Taylor Claire and Christian Collberg. Getting revenge: A system for analyzing reverse engineering behavior. 2019.

[16] Edward Cutrell and Zhiwei Guan. What are you looking for? an eye-tracking study of information usage in web search. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2007.

[17] DARPA. Darpa celebrates cyber grand challenge winners. https://www.darpa.mil/news-events/2016-08-05a, Accessed September 28, 2021.

[18] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. Just-in-time static analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017.

[19] Lev Finkelstein and Shaul Markovitch. Learning to play chess selectively by acquiring move patterns. *ICGA Journal*, 1998.

[20] Vikki Fix, Susan Wiedenbeck, and Jean Scholtz. Mental representations of programs by novices and experts. In *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*, 1993.

[21] Leo Gugerty and Gary Olson. Debugging by skilled and novice programmers. In *Proceedings of the SIGCHI conference on human factors in computing systems*, 1986.

[22] J-M Hoc. *Psychology of programming*. Academic Press, 2014.

[23] Allyson L Holbrook, Melanie C Green, and Jon A Krosnick. Telephone versus face-to-face interviewing of national probability samples with long questionnaires: Comparisons of respondent satisficing and social desirability response bias. *Public opinion quarterly*, 2003.

[24] Alan Jaffe, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, and Bogdan Vasilescu. Meaningful variable names for decompiled code: A machine translation approach. In *Proceedings of the 26th Conference on Program Comprehension*, 2018.

[25] Joby James, L Sandhya, and Ciza Thomas. Detection of phishing urls using machine learning techniques. In *ICCC*. IEEE, 2013.

[26] Anthony R Jansen, Alan F Blackwell, and Kim Marriott. A tool for tracking visual attention: The restricted focus viewer. *Behavior research methods, instruments, & computers*, 2003.

[27] Victor Kaptelinin. Activity theory: Implications for human-computer interaction. *Context and consciousness: Activity theory and human-computer interaction*, 1996.

[28] Andrew J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering*, 2006.

[29] Thomas D LaToza, David Garlan, James D Herbsleb, and Brad A Myers. Program comprehension as fact finding. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007.

[30] Stanley Letovsky. Cognitive processes in program comprehension. *Journal of Systems and software*, 1987.

[31] Robert Levinson and Richard Snyder. Adaptive pattern-oriented chess. In *Machine Learning Proceedings 1991*. Elsevier, 1991.

[32] David C Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. Mental models and software maintenance. *Journal of Systems and Software*, 1987.

[33] Dastyni Loksa, Andrew J Ko, Will Jernigan, Alannah Oleson, Christopher J Mendez, and Margaret M Burnett. Programming, problem solving, and self-awareness: effects of explicit guidance. In *CHI Conference on Human Factors in Computing Systems*, 2016.

[34] Philip Menard, Gregory J Bott, and Robert E Crossler. User motivations in protecting information security: Protection motivation theory versus self-determination theory. *Journal of Management Information Systems*, 2017.

[35] Nikola Milosevic, Ali Dehghantanha, and Kim-Kwang Raymond Choo. Machine learning aided android malware classification. *Computers & Electrical Engineering*, 61, 2017.

[36] Daisuke Miyamoto, Hiroaki Hazeyama, and Youki Kadobayashi. An evaluation of machine learning-based methods for detection of phishing sites. In *International Conference on Neural Information Processing*. Springer, 2008.

[37] Mete Ozay, Inaki Esnaola, Fatos Tunay Yarman Vural, Sanjeev R Kulkarni, and H Vincent Poor. Machine learning methods for attack detection in the smart grid. *IEEE transactions on neural networks and learning systems*.

[38] Bing Pan, Helene A Hembrooke, Geri K Gay, Laura A Granka, Matthew K Feusner, and Jill K Newman. The determinants of web page viewing behavior: an eye-tracking study. In *Proceedings of the 2004 symposium on Eye tracking research & applications*, 2004.

[39] Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. Program comprehension and code complexity metrics: An fmri study. In *ICSE*. IEEE, 2021.

[40] Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology*, 1987.

[41] Radu S Pirscoveanu, Steven S Hansen, Thor MT Larsen, Matija Stevanovic, Jens Myrup Pedersen, and Alexandre Czech. Analysis of malware behavior: Type classification using machine learning. In *CyberSA*. IEEE, 2015.

[42] Keith Rayner, Barbara J. Juhasz, and Alexander Pollatsek. *Eye Movements During Reading*. John Wiley and Sons, Ltd, 2008.

[43] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. How do professional developers comprehend software? In *34th ICSE*. IEEE, 2012.

[44] Bonita Sharif, Michael Falcone, and Jonathan I Maletic. An eye-tracking study on the role of scan time in finding source code defects. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, 2012.

[45] Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna. Rise of the hacrs: Augmenting autonomous cyber reasoning systems with human assistance. In *ACM SIGSAC CCS*, 2017.

[46] Iain Sutherland, George E Kalb, Andrew Blyth, and Gaius Mulley. An empirical examination of the reverse engineering process for binary files. *Computers & Security*, 2006.

[47] Roger Tourangeau and Ting Yan. Sensitive questions in surveys. *Psychological bulletin*, 2007.

[48] Chih-Fong Tsai, Yu-Feng Hsu, Chia-Ying Lin, and Wei-Yang Lin. Intrusion detection by machine learning: A review. *expert systems with applications*, 2009.

[49] Rachel Turner, Michael Falcone, Bonita Sharif, and Alina Lazar. An eye-tracking study assessing the comprehension of c++ and python source code. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, 2014.

[50] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. Survey of machine learning techniques for malware analysis. *Computers & Security*, 81, 2019.

[51] Hidetake Uwano, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto. Analyzing individual performance of source code review using reviewers' eye movement. In *Proceedings of the 2006 symposium on Eye tracking research & applications*, 2006.

[52] Alessio Viticchie, Leonardo Regano, Cataldo Basile, Marco Torchiano, Mariano Ceccato, and Paolo Tonella. Empirical assessment of the effort needed to attack programs protected with client/server code splitting. *Empirical Software Engineering*, 2020.

[53] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S Foster, and Michelle L Mazurek. An observational investigation of reverse engineers' processes. In *29th {USENIX}*, 2020.

[54] Daniel Votipka, Rock Stevens, Elissa Redmiles, Jeremy Hu, and Michelle Mazurek. Hackers vs. testers: A comparison of software vulnerability discovery processes. In *IEEE SP*. IEEE, 2018.

[55] Andrew J Waters, Geoffrey Underwood, and John M Findlay. Studying expertise in music reading: Use of a pattern-matching paradigm. *Perception & psychophysics*, 59(4), 1997.

[56] Susan Wiedenbeck, Vikki Fix, and Jean Scholtz. Characteristics of the mental representations of novice and expert programmers: an empirical study. *International Journal of Man-Machine Studies*, 1993.

[57] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *NDSS*. Citeseer, 2015.

[58] Dennis Yurichev. Reverse engineering for beginners, 2013.

[59] Marwane Zekri, Said El Kafhali, Noureddine Aboutabit, and Youssef Saadi. Ddos attack detection using machine learning techniques in cloud computing environments. In *3rd CloudTech*. IEEE, 2017.

Appendix

A Text of the invitation email

Experiment purpose

A study about how humans coming from different backgrounds and expertise levels (from the 'noob' to 'expert') perform the process of Reverse Engineering and which are the main differences between these categories.

Before starting

The test is completely anonymous, the registration is mandatory but it is quick (just a self-evaluation question). The system will give you a token which is needed for the login, so please preserve it until the end of the test.

The test

For the test, you can find our web-UI at this link (https://reverse.s3.eurecom.fr): it supports some of the main features for RE (commenting code, rename, Xref, ...). After accessing it, the first page comes with a further description of the experiment and of the interface (we invite you to read for the details) and with a list of 3 challenges that you have to solve with our web-UI. The first challenge ('Warmup') is just a warmup one so it is optional and we created it just to make the user become more familiar with the tool. The second and the third challs (namely 'Test 1' and 'Test 2') represent the core part of the experiment. Clicking on one of the two tests starts the RE interface. From now on, your job consists of understanding what the binary does and then submitting a solution in the proper form. You can solve the 2 tests in separate moments and you can stop a RE session and then re-start it (even if we think the best thing is that you stop the RE session after submitting a solution).

Submitting a solution

Note that for the two tests (Test 1 and Test 2), the solution is not required in a specific format (like the flags in a CTF), but it is supposed to be a short description in your own words (just 1 or 2 lines) about the needed steps that make the binary to print the string 'Congratulations' or 'Success!'. Alternatively, also a command line that triggers the correct path in the binary is fine.

Notes

- The interface is not supposed to be a new competitive product, but it is just a tool for the data collection. This does NOT aim to be a "realistic scenario", but a "scenario for which we capture some interesting data". It's an experiment! Please bear with it :-)
- The experience could result a bit painful because basic blocks are blurred when the 'onmouseover' event is captured on another BB. Although we fully understand this makes the RE process slower, this is needed for some aspects we are trying to collect. So, yes, this is NOT your IDA experience you are looking for... it's an experiment! Please bear with it #2 :-)
- For the tests ('Test 1', and 'Test 2'), we disabled the 'Strings' view. Also in this case, the reason is linked with our models and the data we need to collect. So do not worry if you cannot access the 'Strings' view, there is no bug, it is just a design choice.
- In general, we are interested in static analysis, not dynamic one. This explains why we did not add a debugger to the tool. If you are used to reverse with a debugger, that's good! But for this experiment we are interested to know how you would approach a purely static analysis task!
- There is no ranking or prize, this is just an experiment: so please do not cheat.

Thanks a lot for your time/help!

Table 6: Median Time Per Functions for Task 1

Function	BB	BB_{good}	Experts mins (%)	Novices mins (%)	Time Ratio Time Ratio
main	23	17	14.6 (44.6%)	65.3 (45.4%)	x4.4
bridge	12	9	11.4 (34.9%)	52.3 (36.3%)	x4.5
target	3	3	2.4 (6.3%)	8.2 (5.7%)	x3.4
useless-0	1	0	0.17 (0.5%)	0.60 (0.42%)	x3.5
useless-1	4	0	0.58 (1.8%)	2.07 (1.4%)	x3.5
useless-2	16	0	3.5 (10.8%)	15.4 (10.7%)	x4.4
TOTAL	59	29	32.6 (100%)	143.8 (100%)	x4.4

Table 7: Additional hypothesis (not discussed elsewhere)

Hyphothesis	p-value exp	p-value nov
First quartile of time spent on a BB and BB length	0.1	0.6
Interquartile of time spent on a BB and BB length	0.06	0.07
Average of time spent on a BB and BB length	0.08	0.1
Mode of time spent on a BB and BB length	0.4	0.7
T_{first} and T_{max}	0.3	0.2
T_{first} and T_{tot}	0.8	0.8
Solution time and number of BBs she skimmed (i.e., she did a quick look at BB and then a longer one to the same BB)	0.4	0.5
Solution time and how many times the user went back to the previous BB instead of going forward	0.1	0.2
Glanges (i.e., visits of less than 2 seconds) and BB length	0.1	0.1
Solution time and how many times she went to a true branch	0.8	0.1
Solution time and how many times she went to a close branch	0.3	0.2

Table 8: Individual Experts Features

User	Solution Time	Function Exploration (Test1 ; Test2)	Transitions	Tfirst=Ttot	Tfirst=Tmax	Skipped BB
Exp.1	169	Forward,Hybrid;Forward,DFS	2997	16.7%	14.8%	18
Exp.2	137	Forward;Hybrid;Forward,BFS	2551	15.4%	8.3%	19
Exp.3	120	Forward,BFS;Forward,DFS	1662	29.6%	12.9%	24
Exp.4	120	Backward,Hybrid;Backward,BFS	978	29.8%	16.7%	30
Exp.5	163	Forward,Hybrid;Forward,BFS	1654	35.6%	17.2%	4
Exp.6	137	Forward,Hybrid;Forward,Hybrid	2359	18.7%	14.1%	15
Exp.7	134	Forward,Hybrid;Forward,DFS	2845	22.5%	8.3%	21
Exp.8	119	Forward,Hybrid;Forward,Hybrid	1750	29.6%	7.7%	30
Exp.9	156	Forward,Hybrid;Forward,Hybrid	2070	13.5%	21.2%	4
Exp.10	162	Backward,Hybrid;Backward,Hybrid	2141	25.1%	9.6%	24
Exp.11	37	Forward,Hybrid;Forward,Hybrid	450	46.4%	14.8%	31
Exp.12	34	Backward,Hybrid;Backward,Hybrid	727	35.4%	16.7%	25
Exp.13	118	Forward,BFS;Backward;Hybrid	1546	9.6%	18.7%	2
Exp.14	119	Forward,BFS;Forward,BFS	1842	11.6%	14.8%	4
Exp.15	96	Forward,BFS;Forward,BFS	1564	19.3%	17.4%	23
Exp.16	154	Forward,Hybrid;Forward,BFS	2547	10.9%	14.8%	1
Exp.17	80	Forward,BFS;Forward,DFS	1321	23.8%	16.7%	13
Exp.18	48	Forward,Hybrid;Forward,BFS	761	34.8%	10.9%	41
Exp.19	81	Forward,BFS;Backward,BFS	746	40.0%	13.8%	3
Exp.20	48	Forward,Hybrid;Forward,DFS	818	22.5%	21.2%	34
Exp.21	38	Forward,BFS;Forward,BFS	483	47.7%	6.4%	27
Exp.22	43	Backward,DFS;Backward,Hybrid	627	41.2%	20.0%	29
Exp.23	44	Forward,Hybrid;Forward,Hybrid	673	39.3%	16.7%	32
Exp.24	27	Backward,Hybrid;Backward,DFS	462	46.4%	20.6%	30
Exp.25	29	Forward,Hybrid;Forward,Hybrid	634	36.1%	23.8%	45
Exp.26	45	Forward,Hybrid;Forward,Hybrid	789	35.4%	18.0%	27
Exp.27	64	Forward,Hybrid;Forward,Hybrid	835	45.1%	11.6%	21
Exp.28	70	Forward,Hybrid;Forward,BFS	1478	31.6%	10.9%	24
Exp.29	32	Forward,Hybrid;Forward,Hybrid	820	38.7%	14.1%	23
Exp.30	89	Backward,DFS;Backward,Hybrid	1415	14.1%	15.4%	8
Exp.31	171	Forward,Hybrid;Forward,BFS	2115	10.9%	10.9%	26
Exp.32	56	Forward,BFS;Forward,BFS	517	45.8%	14.8%	44
Exp.33	106	Forward,Hybrid;Forward,Hybrid	988	38.0%	17.4%	24
Nov.1	266	Forward,Hybrid;Backward,Hybrid	3330	5.8%	10.3%	2
Nov.2	329	Forward,Hybrid;Forward,BFS	5114	7.7%	6.4%	7
Nov.3	304	Forward,BFS;Forward,BFS	5025	14.1%	8.3%	13
Nov.4	208	Forward,Hybrid;Forward,BFS	3793	12.9%	9.0%	9
Nov.5	260	Forward,Hybrid;Forward,Hybrid	3560	20.0%	5.1%	19
Nov.6	257	Forward,Hybrid;Forward,DFS	4136	8.3%	8.3%	4
Nov.7	264	Forward,Hybrid;Backward,Hybrid	3433	5.1%	9.0%	1
Nov.8	331	Forward,Hybrid;Forward,Hybrid	3805	3.2%	10.9%	0
Nov.9	303	Forward,Hybrid;Forward,Hybrid	3892	16.1%	9.6%	20
Nov.10	415	Forward,Hybrid;Forward,BFS	7602	12.2%	3.8%	10
Nov.11	371	Forward,BFS;Forward,BFS	4796	9.6%	10.3%	8
Nov.12	381	Forward,Hybrid;Backward,Hybrid	4514	16.1%	9.1%	17
Nov.13	258	Forward,Hybrid;Forward,BFS	2374	1.2%	10.3%	0
Nov.14	458	Sequential,Hybrid;Sequential,Hybrid	6955	4.5%	6.4%	1
Nov.15	251	Forward,Hybrid;Backward,BFS	3067	24.5%	14.8%	22
Nov.16	409	Forward,BFS;Forward,BFS	5656	4.5%	9.6%	2
Nov.17	481	Forward,BFS;Sequential,BFS	6270	8.3%	4.5%	9
Nov.18	560	Backward,Hybrid;Sequential,Hybrid	6976	5.1%	7.1%	1
Nov.19	194	Forward,Hybrid;Forward,Hybrid	2791	7.7%	13.5%	1
Nov.20	351	Forward,Hybrid;Forward,Hybrid	3685	4.5%	10.3%	1
Nov.21	301	Forward,Hybrid;Forward,Hybrid	5020	10.3%	10.9%	3
Nov.22	228	Backward,BFS;Backward,Hybrid	2841	16.1%	12.2%	19
Nov.23	300	Forward,BFS;Forward,Hybrid	3091	7.7%	12.3%	6
Nov.24	195	Forward,Hybrid;Forward,Hybrid	2592	6.4%	14.1%	0
Nov.25	261	Forward,Hybrid;Forward,BFS	2510	11.6%	13.5%	3
Nov.26	240	Forward,BFS;Forward,BFS	3050	15.4%	9.6%	11
Nov.27	740	Sequential,Hybrid;Sequential,Hybrid	7879	6.4%	13.1%	8
Nov.28	543	Forward,Hybrid;Sequential,Hybrid	6641	2.5%	7.7%	0
Nov.29	941	Sequential,Hybrid;Sequential,Hybrid	8150	0.0%	1.9%	0
Nov.30	320	Forward,BFS;Forward,BFS	2912	12.5%	7.1%	16
Nov.31	316	Forward,Hybrid;Forward,BFS	2886	18.7%	10.9%	13
Nov.32	234	Forward,Hybrid;Forward,BFS	4048	7.7%	6.4%	3
Nov.33	181	Forward,BFS;Forward,BFS	3445	12.2%	8.3%	10
Nov.34	207	Forward,Hybrid;Forward,Hybrid	2202	29.0%	9.6%	26
Nov.35	513	Forward,BFS;Sequential;Forward,BFS	6233	0.0%	4.5%	0
Nov.36	199	Forward,Hybrid;Forward,Hybrid	3408	18.7%	6.4%	17
Nov.37	178	Forward,BFS;Forward,Hybrid	1618	5.8%	19.3%	3
Nov.38	441	Forward,Hybrid;Sequential,Hybrid	5946	6.4%	8.3%	1
Nov.39	253	Forward,Hybrid;Forward,Hybrid	3486	21.9	10.9%	7