DECOMPERSON: How Humans Decompile and What We Can Learn From It

Kevin Burk
UC Santa Barbara
kburk@ucsb.edu

Fabio Pagani
UC Santa Barbara
pagani@ucsb.edu

Christopher Kruegel
UC Santa Barbara
chris@cs.ucsb.edu

Giovanni Vigna
UC Santa Barbara
vigna@cs.ucsb.edu

Abstract

Human analysts must reverse engineer binary programs as a prerequisite for a number of security tasks, such as vulnerability analysis, malware detection, and firmware re-hosting. Existing studies of human reversers and the processes they follow are limited in size and often use qualitative metrics that require subjective evaluation.

In this paper, we reframe the problem of reverse engineering binaries as the problem of *perfect decompilation*, which is the process of recovering, from a binary program, source code that, when compiled, produces binary code that is identical to the original binary. This gives us a quantitative measure of understanding, and lets us examine the reversing process programmatically.

We developed a tool, called DECOMPERSON, that supported a group of reverse engineers during a large-scale security competition designed to collect information about the participants' reverse engineering process, with the well-defined goal of achieving perfect decompilation. Over 150 people participated, and we collected more than 35,000 code submissions, the largest manual reverse engineering dataset to date. This includes snapshots of over 300 successful perfect decompilation attempts. In this paper, we show how perfect decompilation allows programmatic analysis of such large datasets, providing new insights into the reverse engineering process.

1 Introduction

The reverse engineering of binary code is a key process in a number of security tasks, from malware analysis to vulnerability discovery. Unfortunately, reverse engineering binary code is a challenging task, as the process that transforms source code into executable code results, in most cases, in the loss of high-level constructs, such as structured control flow, complex data structures, variable names, and function signatures.

The fundamental task of reverse engineering is to deduce the information that the compiler has obscured, and to report it in a human-friendly format. Reverse engineers often use source code as this format; the process of recreating source code from binary code is known as *decompilation*.

Decompiled code is often incomplete or pseudo-code, and not valid in any real programming language. This is still useful as a guide for humans trying to understand the binary, but we believe that decompilation becomes even more useful when it conforms to a programming language—ideally that of the original program.

In the absolute best case, decompiled code would be valid source code, and re-create the original binary when compiled. This would enable a wide variety of new operations. For example, such decompiled code could be used for patching, allowing security analysts to patch at the source level instead of directly modifying the binary [36]. This code could also be used with source-based analysis techniques [31], or recompiled with the instrumentation required by fuzzing tools. It could also be used to port a binary to other architectures—from x86-64 to ARM, for example—or it could be recompiled with additional optimizations [44].

We assert that this "perfect" decompilation is the ultimate goal of all decompilers, human or otherwise. We show that perfect decompilation is achievable by human reverse engineers today, and that it allows study of the reversing process at scale. These larger studies can then be used to improve the automatic compilers of tomorrow.

Automatic Decompilers. Decompilation is a challenging process, requiring intense focus from highly skilled reverse engineers. Therefore, the ability to decompile binaries programmatically can save a significant amount of time and effort. Even partial decompilation can be useful for program comprehension, as source code is much easier to read than raw assembly.

Automatic decompilation techniques have received much attention in the past few decades. Cifuentes laid the foundations of modern decompilers in the mid-nineties [9], and further research works have proposed techniques to improve the recovery of control-flow constructs [5, 18, 49–51], types [7, 12, 13, 29, 32], and even variable names [1, 2, 7, 21, 22, 26]. More recently, a number of machine learning techniques have also been used to improve decompilation [15, 16, 19, 23, 24, 28].

Most of these approaches use readability metrics for evaluation, as they are intended to help human reverse engineers. Only a few of them focus on the correctness of the decompiled code. Schwartz et al. [5] were the first to propose correctness as a metric to evaluate a decompiler. They use a test suite to determine whether or not a decompiled program preserves the semantics of the original binary. Schulte et al. [38] extended this concept by introducing Byte-Equivalent Decompilation (BED), a decompilation technique based on genetic programming. The authors use byte-similarity between binaries as a fitness function for their evolutionary algorithm; any individual that reaches perfect byte-equivalence is guaranteed to be algorithmically equivalent to the original binary. The topic has also been explored by Verbeek et al. [44], who recently developed FoxDec, a decompilation framework based on formal methods that produces sound and recompilable source code. In this case, the recompiled code is guaranteed to have the correct behavior, but this may be difficult to verify independently, as the resulting binary could be considerably different from the original.

In general, automatic decompilers that aim for readability tend to sacrifice correctness (or at least recompilability), while those that aim for correctness do so at the expense of readability. If we want decompilation that is both correct and human-friendly, we must still rely on the ingenuity of humans.

Human Decompilers. Given that human reverse engineers still produce the best decompiled code, it is surprising that there are very few studies on how they do so. Existing studies focus on higher-level understanding of binary programs, and not on the low-level art of decompilation, which could provide insights that could improve automatic decompilers. This paper aims to bridge that gap.

In general, the problem of reverse engineering binaries falls under the umbrella of program comprehension, which is the study of how programmers understand unfamiliar code bases [27, 40, 45]. This is a very broad field of research, which includes understanding how programmers repair and maintain software [46, 47] and how to create better tools to support the comprehension process [25, 41].

In the past few years, program comprehension researchers have started examining the process of reverse engineering. Bryant [6], for example, presents three studies that explore the conceptual aspects of reversing by various means: notes and video taken during a reversing experiment, a semi-structured interview with experts, and a "think-aloud" session where participants were asked to reverse a key generation algorithm. More recently, Votipka et al. [48] interviewed sixteen reverse engineers as they reenacted a recent reverse engineering activity. These interviews were also semi-structured, and the authors used them to extract common themes. Taylor [42] presented a system to study the behavior of reverse engineers by using an instrumented virtual machine that collects mouse and keyboard events, process creations, and changes in window focus. Finally, Mantovani et al. [30] studied how humans reverse engineer binary code, with an emphasis on the differences between novices and experts. Their approach used a Restricted Focus Viewer: basic blocks were blurred unless selected, allowing the authors to precisely track the subjects' attention. Most of these studies were—necessarily—small. They examined the reversing process using methods that were human-friendly but not amenable to automated analysis. As a result, the data was limited to what the researchers could analyze manually.

Exploring the reversing process at scale remains challenging. The biggest obstacle is the difficulty of quantifying the reversers' understanding: there is currently no metric that can describe how well a reverse engineer understands a function. This is further complicated by the fact that experience plays a key role in reversing, and reversers of different skill levels may reach very different levels of understanding. Even having a standard set of tasks does not guarantee consistency. Expert reversers might be able to carry out their tasks successfully with limited (but focused) understanding, while novices might need to acquire more in-depth, detailed knowledge about a binary before being able to meaningfully modify or exploit it.

We believe that the problem of quantifying comprehension can be addressed by using a more formal language to explain program behavior. Specifically, we require reverse engineers to communicate their understanding of a function in the form of compilable source code. In other words, we compare the process of understanding binary code to the process of decompilation.

2 Perfect Decompilation

In this paper, we study the human reverse engineering process in terms of a new metric of program comprehension that we call *perfect decompilation*. We consider decompiled code for a function to be perfect if, when compiled with the same compiler and options as the original, it produces the exact same assembly.

Benefits. Expecting perfect decompilation is a high bar, but it allows for a number of useful features. Primarily, it gives us a quantitative measure of program comprehension: decompiled code can be evaluated based on how close it is to perfect. In this paper, we measure understanding as the Jaccard similarity coefficient over lines of assembly code.

Perfect decompilation also gives reverse engineers an obvious "done" state. Code that achieves perfect decompilation has very clearly captured the behavior of the original function. It also avoids the ambiguity present in pseudo-code and natural language. Saying "this function averages a list of integers" does not capture what happens when the list is empty, but the source code of the function will.

Furthermore, defining everything in terms of source and assembly code—both common and well-defined representations of programs—allows for reverse engineers' mental models to be analyzed programmatically. As a result, studies can be scaled to hundreds of individuals, unlike interview-based approaches that are necessarily small and focus on qualitative results.

Finally, perfect decompilation is trivial to prove correct: a simple assembly diff is enough to prove perfection (or to show what instructions differ). In contrast, proving that code is functionally equivalent requires unit tests, and even these are insufficient without an additional measure of coverage that ensures all code paths are tested.

Limitations. Perfect decompilation is not a traditional reverse engineering task, so there is some risk that using this metric will distort the reversing process. For example, reverse engineers do not usually report their findings as compilable source code, so this requirement will add more work. However, we believe that reverse engineers already have a code-like mental model of binaries, and that it takes only minor extra work to make this explicit.

Perfect decompilation also focuses on the careful analysis of functions. Other skills are also important for reverse engineers—for example, the ability to skim a large binary

to determine on which functions to focus—but these will not necessarily be captured by our metrics.

There is also a risk that reverse engineers may spend a non-negligible amount of time "fighting with the compiler." This refers to the extra time spent by reversers who have a complete mental model of a function, but struggle to reach perfect decompilation due to the idiosyncrasies of the compiler.

We believe that only the third point above is a true threat to our work, and that it is outweighed by the benefits of using perfect decompilation.

Contributions. We define perfect decompilation as a new, quantifiable metric of program comprehension. We develop a tool, called DECOMPERSON, that supports human reverse engineers during the decompilation process and helps them achieve perfect decompilation, recording the steps they take along the way. We then set out to answer the following research questions:

RQ1: Are humans capable of perfect decompilation?

RQ2: If they are, what processes do they follow when decompiling?

RQ3: Are these processes representative of more traditional reverse engineering (program comprehension) processes?

Our findings are based on the submissions to an online security competition in which reverse engineers were asked to use DECOMPERSON while decompiling binaries written in five different languages.[1] Over 150 people participated—twice as many as the next-largest study, and an order of magnitude more than most—making this, to the best of our knowledge, the largest decompilation study to date. It is also the first study that has a concrete metric for levels of understanding during the reverse engineering process.

By analyzing these submissions, we show that perfect decompilation is within reach of humans today, and therefore a reasonable goal for the automated decompilers of the future. The results also show which key activities performed by humans should be targeted by tools supporting the reverse engineering process, and provide insights that can be used to build and improve automated and semi-automated decompilers.

3 DECOMPERSON

We developed a custom tool, which we called DECOMPERSON, to collect source code submissions. The tool was designed to give reverse engineers complete control over their source code and to encourage frequent submissions, allowing us to collect many "snapshots" over the course of the reversing process.

Typical reversing tools, such as Hex-Rays [20], Ghidra [17], or Binary Ninja [3], provide some form of decompilation functionality,[2] but users have very little control over the output. User input is typically limited to a small set of well-defined interactions, such as renaming variables or functions, providing type annotations, and inserting comments. When using these tools, a reverse engineer's workflow consists of iteratively applying these annotations in order to coax better and better source code out of the decompiler.

We believe that this style of interaction is too restrictive to reliably produce perfect decompilation. As an alternative, DECOMPERSON gives users complete control over the decompiled code and helps them modify it until it compiles identically to the original binary. Since this method of decompilation is heavily reliant on the user, the tool attempts to make the user's job as easy as possible. DECOMPERSON makes discrepancies between the generated binary and the original one obvious, and it provides a tight feedback loop, showing the user how each source code modification changes the generated binary code.

The high-level workflow of DECOMPERSON is the following: ① users edit the source code using our interface; ② the source code is sent to our back-end component, where it is compiled, disassembled, tested, and diffed against the original binary; ③ the binary is scored based on the metrics described in Section 4, and the results, including the diff, are sent back to the interface.

The Interface. The heart of the DECOMPERSON system is the user interface, shown in Figure 1. On the left side, the interface shows editable source code, which the user can compile at any time. On the right side, the interface shows the target disassembly and any output from the compilation process. When compilation fails, the right panel displays any errors reported by the compiler. When compilation succeeds, DECOMPERSON shows the disassembly of the resulting binary, both in raw format and in diff-style, which shows the differences between the compiled binary and the original binary's assembly code.

This diff view is the most important part of the system, as it shows reverse engineers exactly where their binary does not match the target, providing immediate feedback on the effects of any source code change—typically within two seconds of the user hitting the "Compile" button. In order to help reversers focus on the more important structural changes first, minor differences—such as instructions that differ only in their operands—get more subdued highlighting.[3] Both styles of highlighting are visible in Figure 1.

The interface also provides a few other useful features. For example, clicking a line of assembly highlights the corresponding line of source code; this mechanism is responsible for the matching blue highlights in Figure 1. The interface also offers a "Replace" function that lets reversers make cosmetic replacements that clean up the disassembly view; for example, one of our playtesters used this feature to hide stack layout

[1] Most existing studies and associated tools focus on C/C++ decompilation only; we also cover Go, Rust, and Swift.

[2] At the time of writing, these tools are not able to produce decompiled code that can be recompiled.

[3] The majority of these differences are caused by mismatched stack layouts, as seen on lines 33–34 of Figure 1.

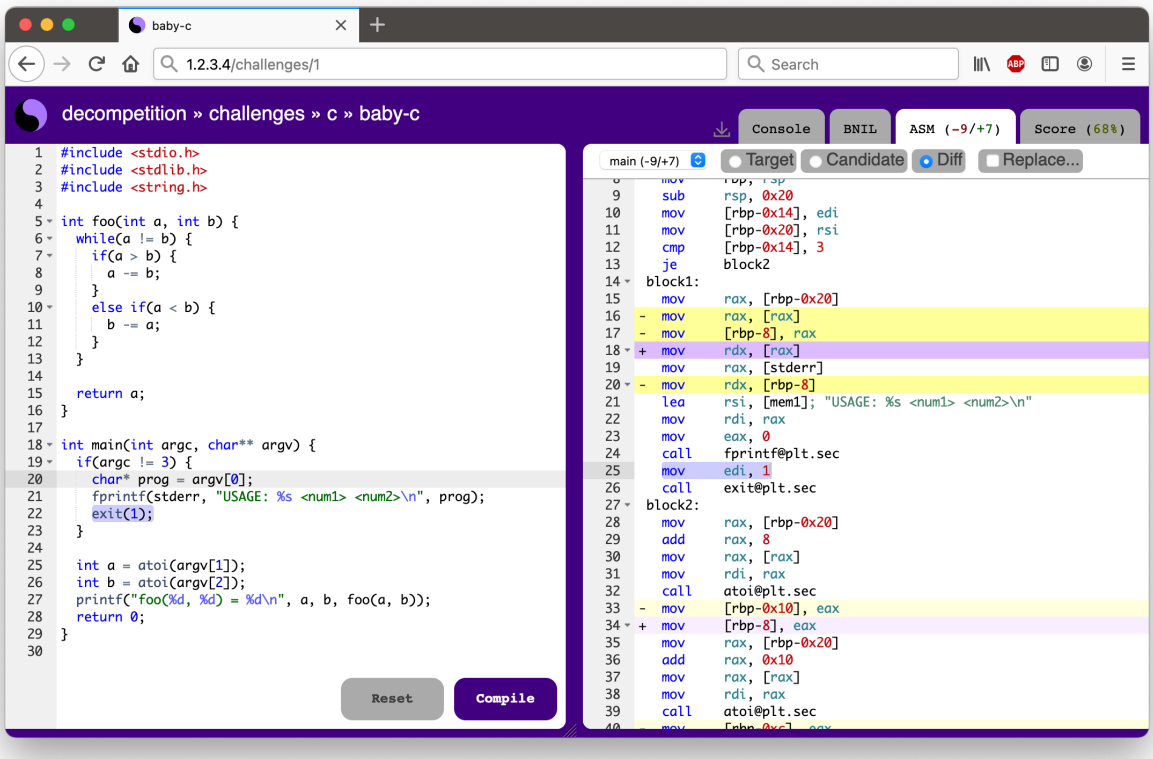

Figure 1: The DECOMPERSON interface showing a partial solution to the `baby-c` challenge. The current decompiled source code (provided by the user) is visible on the left, and the assembly diff is shown on the right. Instructions present in the target but missing from the submission are shown in purple, while instructions that should be removed from the submission are shown in yellow. Lines 16–20 of the disassembly show significant differences, and are displayed with a bright background; lines 33–34 contain only operand differences, and are shown with a subdued background. The user has clicked on line 25 of the assembly code, and the corresponding line of source code (line 22) has been highlighted to match.

mismatches completely—decluttering the diff while working on control flow—and then fixed the stack layout at the end.

The Disassembler. We chose to show binary differences as line-based assembly diffs. This has the benefit of familiarity: all reverse engineers know assembly, and virtually all will have used a command line diffing tool, like `diff` or `git`. There is also the benefit of simplicity: the diff format can be displayed in any text editor.

To keep these diffs usable, we needed assembly that was free of any noise that would lead to spurious highlighting. The main sources of noise are absolute memory addresses, which depend on how the compiler lays out the binary. Examples of these addresses are code references (such as addresses used in `call` and `jmp` instructions) and global data references (such as addresses in the `.bss` region). Highlighting diffs that include these references would be cumbersome for DECOMPERSON's users, since these differences are rarely fixable by editing the source code of the function in which they appear. To avoid these problems, DECOMPERSON includes a custom disassembler that replaces absolute memory references with symbols where possible, or generic identifiers where not. For example, lines 13–14 of the assembly code in Figure 1 show a sanitized block label and jump, line 19 shows a symbol replacement, and line 21 shows a sanitized global memory reference. The disassembler also recovers string constants, as seen on line 21, helping reversers by keeping the disassembly as self-contained as possible.

The Back-End. Whenever a user hits the "Compile" button, the user-provided source code is sent to the DECOMPERSON back-end. This component compiles and tests the source code submission in a container, and then scores it based on the differences between its assembly and that of the target. If unit tests are available for the binary, the submission is also scored based on its performance on the tests. These scores are used as a measure of source code correctness, and also serve as the basis of the competition described in Section 4.

Compiling submissions on a central server allows us to use the correct compiler for each submission, and also relieves users of the burden of installing and managing

different compilers for each language. In the current version of DECOMPERSON, we presume to know the toolchain and the compilation options used to build the original binaries, and use the correct versions to compile any user submissions. This was sufficient for our research; a more advanced version could integrate compiler provenance techniques to automatically extract this information from binaries [8, 33–35, 37] and allow users to add or select new compilers. This architecture also allowed us to collect every source code submission that our users made. This generated a large corpus of submissions, recording our users' reverse engineering processes from their very first tentative solutions to an exact disassembly match.

4 DECOMPETITION

The data we analyze in this paper comes from an online competition called DECOMPETITION, which we designed and ran in November 2020. This was a competition purely focused on reverse engineering and, to the best of our knowledge, the first of its kind.[4]

Format. DECOMPETITION was a 24-hour Jeopardy-style competition.[5] Each challenge was an executable binary, and the competitors' goal was to write source code that would compile to an identical binary. Scores were based on binary similarity, and calculated by both assembly diffing and unit testing.

Competitors were required to submit their source code through the DECOMPERSON interface, but were not required to use the interface for reversing; output from other reversing tools could easily be pasted in. Competitors were given access to DECOMPERSON and all the challenges as soon as the competition started, and could make any number of submissions.

Competitors played on teams of unlimited size. Anyone wishing to play solo could play as a team of one, and most players chose to do so.

Challenges. Multiple people from the UCSB security lab contributed a total of 23 challenge binaries, covering five major compiled systems languages: C, C++, Go, Rust, and Swift. The binaries were not stripped, but did not include any debug information. An overview of these challenges is shown in Table 1.

The challenges were designed to use the features and peculiarities of each programming language. For instance, a challenge written in C traversed a linked list of structs, while a C++ challenge made heavy use of classes and Standard Template Library containers. Similarly, a challenge written in Go was designed to use complex numbers, while another challenge made use of Rust traits and networking.

We selected challenge ideas primarily to cover a variety of language features. A spectrum of difficulty levels was also desirable, but harder to estimate; challenge authors controlled difficulty by limiting the number of lines of source code. Authors also provided some scaffolding for each challenge:

- A set of unit tests to ensure that the semantics of the original program were reproduced.
- Starter code to provide to the DECOMPETITION participants, which typically contained only stubs for the functions a participant was expected to decompile, data-type stubs, or include directives.
- A list of the functions that should be disassembled and diffed, which was used as an input to the disassembler, to make sure it only disassembled user-defined functions.

As a bonus, we provided Binary Ninja Intermediate Language [4] for all challenges. This is a "semantic representation of [...] assembly language instructions"—a form of decompilation that attempts to capture the operation of binary code without tying it to any particular programming language. This intermediate language served as an additional perspective on the challenge binaries, and could be used even by participants who did not have access to a decompiler. Like other forms of decompilation, it tended to be helpful for C binaries, but less so for other languages.

Scoring. We assigned each challenge an arbitrary value between 100 and 500 points. These values reflected estimated difficulty, and were mainly based on challenge size and the presence of language-specific constructs that would make the binary harder to reverse.

The values are well-correlated with the number of assembly lines per challenge (Pearson's $r = 0.83$), and their inverses are well-correlated with the number of perfect submissions per challenge ($r = 0.70$).[6] There is also a correlation between challenge value and average time to solve, but this is weaker ($r = 0.54$ for mean; $r = 0.59$ for median).

One challenge per language was designated as a "baby" challenge. These challenges were simple but worth extra points, encouraging players to start there to learn the basics of reversing code written in that specific language and to familiarize themselves with DECOMPERSON.

Each submission received a score based on the following weighted components:

Test Score: Each challenge had a collection of test cases that were not disclosed to the participants. Test case points were awarded proportional to the number of test cases passed. These points made up 20% of the total score.

Diff Score: The majority of the points came from matching the target assembly. Each submission was compiled and its user-implemented functions were disassembled. This

[4] This research has received IRB scrutiny, and was determined to be exempt under Category 3. No personally identifiable information was used.

[5] In Jeopardy-style security competitions, participants solve a number of challenges from a list of options. They do not interact with other participants, as is instead customary in traditional attack-defense security competitions. Both Jeopardy-style and attack-defense competitions are often referred to as "Capture the Flag" (CTF) competitions.

[6] This jumps to $r = 0.90$ if the 200-point `baby-c` challenge is considered an outlier and removed.

	Challenge Statistics					Submission Statistics			
Language	Challs	Points	Source Lines	Assembly Lines	Ratio	Submissions	Compilable	Passing Tests	Perfect Decompilation
C	4	500	110	396	3.60	10,060	8,022	1,634	186
C++	5	1,400	491	2,548	5.19	9,155	6,065	1,091	50
Go	7	1,800	310	1,730	5.58	10,192	7,617	4,018	44
Rust	4	1,300	219	1,695	7.74	3,524	2,561	1,252	26
Swift	3	1,200	178	2,871	16.13	2,599	1,686	405	23
Total	23	6,200	1,486	9,240	7.06	35,530	25,951	8,400	329

Table 1: An overview of the DECOMPETITION challenges and submissions. *Ratio* is the average number of assembly lines produced per source line. Only one perfect submission is counted per (team, challenge) pair. The number of Go submissions is inflated by the top two teams trying to brute-force the unsolved `julie` challenge. See Appendix C for a more detailed breakdown.

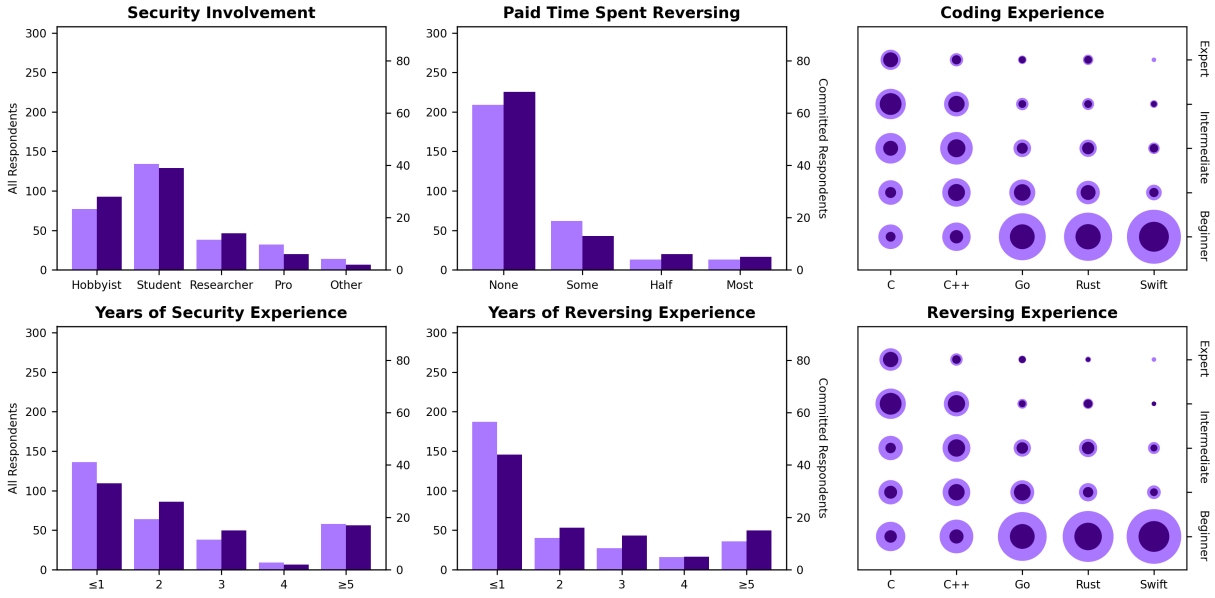

Figure 2: Demographic information from the background survey. The bar graphs on the left show how participants are involved in computer security, and for how long. The area plots on the right show how familiar participants are with programming and reversing each of the languages used in the competition, as measured on a five-point Likert scale. Responses from all 308 respondents are shown in light purple; responses from the 93 "committed" respondents who earned a 50% score or better on at least one challenge are shown in dark purple.

portion of the score was calculated as the intersection-over-union (Jaccard similarity coefficient) of the submission versus the target, and made up 60% of the total score.

Bonus Score: In order to encourage participants to generate perfect decompilation, the final 20% of the points were awarded only when a submission matched the target assembly exactly.

A team's score on a challenge was the maximum score achieved by any of its members' submissions. A team's total score was the sum of these per-challenge scores. There was no limit on the number or frequency of submissions, and any new record would improve the team's score; there was no way for a team to lose points.

Recruiting. We recruited participants by word-of-mouth and by announcing the competition on CTFtime [11], a popular online index of security and hacking competitions. To encourage participation, we split a $1,000 prize pool among the top three teams and provided Binary Ninja licenses [3] to the top two teams.

A total of 425 users and 238 teams registered for DECOMPETITION, but only a fraction of those played. A total of 188 users and 139 teams made source code submissions; 159 users and 114 teams submitted source code that could be compiled.

Demographics. Before the competition, we invited the players to fill out a survey about their involvement in reverse engineering. This survey consisted mainly of questions about how—and for how long—the participants were involved in computer security. The full survey can be found in Appendix A.

A total of 308 people (72% of all registrants) filled out this survey; the results are summarized in Figure 2. The large majority of participants were either students or hobbyists, and only a few worked in a position where they were paid as reverse engineers. The majority were also new to computer security, and to reverse engineering in particular—the number of players at each year of experience decreases exponentially.

The survey also asked participants how familiar they were with programming and reversing the languages used for challenges. We found that participants were well versed in C and C++ but significantly less familiar with Go, Rust, and Swift. Given that these latter languages are becoming more popular systems languages, this lack of familiarity may indicate a gap in security education or a lack of tooling support.

We asked players to fill out a similar survey after the competition. This follow-up survey is described in Appendix B, and was intended to capture any changes in players' perception of reversing that occurred during the competition; it also provided a place to leave general feedback. Only 49 players (12% of all registrants) chose to complete this second survey, so it is likely less representative than the first.

Submissions. In order to study the process that humans use when decompiling, we recorded all submissions made during DECOMPETITION. As summarized in Table 1, we received a total of 35,530 source code submissions, 25,951 (73%) of which successfully compiled. The majority of the submissions were for challenges written in C, C++, and Go. With about 10,000 submissions each, these three languages account for almost 83% of all submissions. Rust and Swift were less popular, with around 3,000 submissions per language, likely because of their perceived difficulty.

While fewer experts played than beginners, these experts were able to solve more challenges, and thus made more submissions. Overall, the submissions were balanced across different levels of expertise. In total, 32,254 submissions (90% of all submissions) were made by users who completed the background survey. Among these, we received 9,553 submissions from expert users who claimed five or more years of security experience, 7,155 (75%) of which compiled; 12,963 submissions from players with 2 to 4 years of experience, 9,271 (72%) of which compiled; and 9,738 submissions from beginners with less than 2 years of experience, 6,908 (71%) of which compiled. We estimate that beginners submitted around 30% of all decompilation attempts and intermediate players around 40%, with experts responsible for the remaining 30%.

The number of submissions that passed all unit tests is also interesting, but harder to interpret. Perhaps counterintuitively, a language or challenge that has a higher percentage of passing submissions is likely harder to perfectly decompile—the reversers were able to replicate the binary's behavior early in the reversing process, but then had to make many more submissions while trying to fix mismatches caused by the compiler's handling of various language constructs. Reversing style also affects this statistic: reversers who reverse one function at a time cannot pass all tests until they start work on the final function.

There are also some confounding factors. For example, all participants worked on the C challenges, including those who never made much progress, while only the most committed reversers dared attempt the Rust and Swift challenges. Additionally, the number of Go submissions was inflated by the top two teams trying to brute-force the unsolved `julie` challenge.

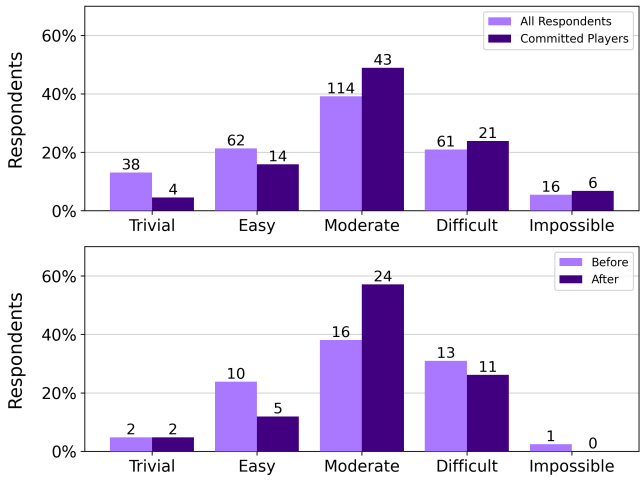

Figure 3: How difficult is completely recreating the source code of a small binary? Top: Survey responses from all 308 respondents compared with those from the 93 committed players who achieved a 50% score or better on at least one challenge. Bottom: Responses before and after the competition from the 42 players who answered this question on both surveys.

5 RQ1: Can Humans Perfectly Decompile?

Expectations. We started this project unsure whether or not humans would be able to consistently perform perfect decompilation. It was interesting, then, that many DECOMPETITION participants did not share our conservative views.

One of the questions in our pre-competition survey asked the participants to rate, on a scale from trivial to impossible, how difficult they thought it would be to recreate the source code of a small binary. We summarize the responses in Figure 3. The majority (37%) of all respondents thought it would be a moderately difficult task, and 12% claimed that it would be trivially easy. Only 20% of the responders thought it would be difficult, and 5% thought it would be impossible.

The players who were able to reach a 50% score on at least one challenge, however, had more realistic expectations. Only 4% of this group had originally thought that perfect decompilation would be trivially achievable,[7] 23% thought

[7] Of the 19 "trivial" respondents who made source code submissions, 15 gave up without reaching a 50% score on any challenge.

	Reg'd	0%	>0%	20%	40%	60%	80%	100%
Users	425	188	159	120	110	103	91	91
Teams	238	139	114	82	76	72	66	66

Table 2: Numbers of users and teams who registered and who made any submission reaching the listed cutoff scores on any challenge. All submissions that successfully compiled got a score greater than zero. The lack of attrition between 80% and 100% is an artifact of the 20% perfect match bonus described in Section 4: any submission reaching 80% would earn the bonus and jump up to a perfect score.

it would be difficult, and the majority (46%) believed it to be a moderately difficult goal.

It is also interesting to see how opinions changed after using DECOMPERSON. Out of the 42 respondents that answered both surveys, 11 lowered their difficulty expectations, 10 increased them, and 21 did not revise their assessment. Overall, the trend was towards a tighter clustering around "moderate" difficulty, and after the competition, none of the respondents thought that perfectly decompiling a binary was impossible.

Perfect Decompilation. The results of DECOMPETITION show that the participants' optimism was justified. Generating perfect decompilation was certainly not trivial, but of the players who were willing to put in the effort, a surprisingly large number were able to achieve this goal, even when confronted with higher-level languages.

As shown in the last column of Table 2, a total of 91 players reached a perfect score on at least one challenge. This is only 48% of the 188 users who submitted source code, but 57% of the 159 users who persisted until their code compiled. This figure only improves if we set a higher bar for persistence: of the 120 players who earned a 20% score on any challenge, for instance, more than 75% would go on to reach a perfect score on at least one challenge.

As reported in Table 1, the language with the highest number of perfect submissions was C (186 submissions), followed by C++ (50), Go (44), Rust (26), and finally Swift (23). Interestingly, this order perfectly follows the languages' ratio order reported in Table 1—higher-level languages tend to produce more lines of assembly per source line, and are also more difficult to decompile.

All challenges were attempted, and all but one were perfectly decompiled by at least one participant. The top two teams were able to solve all challenges except one—the Go challenge `julie`—and both teams were able to pass all unit tests and reach at least a 90% diff score on this unsolved challenge.[8] Furthermore, with the exception of the C++ challenge `lambic`, where lambda expressions were used extensively, all other challenges received at least three perfect solutions.

Player Distribution. Figure 4 gives a breakdown of the players who made any submission earning at least a 20% score, categorized both by years of experience and by security involvement. Solves per player are broken down into C and non-C plots, as players treated C challenges differently than the rest.

Virtually all players attempted the C challenges, and even the least-solved C challenge—the `rootkit` challenge, which built and traversed a singly linked list of user-defined structs—received 30 perfect solutions; the `baby-c` challenge received the most, with 66. This resulted in a high number of perfect solutions per player, even though there were only four C challenges available. Experience had some effect, but not a large one; beginners are able to reverse C relatively well.

On the other hand, experience played a more significant role when reversing the other languages. As seen in Figure 4, the Solves per Player ratio drops dramatically for players with one or fewer years of experience; only a small number of beginners submitted perfect solutions, despite there being many more non-C challenges. Challenges written in these languages also received more solutions from participants with longer security careers, and those with five or more years of reversing experience had a clear advantage. There was also a more marked difference based on role, with researchers outperforming the other participants.

Summary. These results validate our potentially ambitious claims from Section 1: perfect decompilation is a reasonable standard of program comprehension, reachable even by non-experts. Experience, however, is certainly helpful, especially when reversing higher-level languages.

6 RQ2: How do Humans Decompile?

Having shown that humans are capable of producing perfect decompilation, the next step is to investigate how they do it.

Understanding this process is beneficial for a number of different applications. In particular, a better grasp of the reversing process and identifying the roadblocks that stand in the way of perfect decompilation will support the design and implementation of better tools to support human reversers. This knowledge may also help improve automatic decompilers, which can incorporate insights learned from human experts.

Since we collected tens of thousands of source code submissions, we were interested in analyses that could be performed programmatically. The metrics described below provide quantitative alternatives to the more traditional interview method of studying the reversing process.

Code and Score Changes. The first analysis we performed involved the magnitude of the players' code changes and

[8] The `julie` challenge was a Go program that used a syscall to get the terminal dimensions, then used `complex128`s to calculate and draw a Julia set as ASCII art. The first-place team had explicitly zero-initialized their `winsize` struct and captured it as a struct variable rather than a reference, both changes that (for some reason) caused Go's register allocator to behave differently. The second-place team handled the struct correctly, but did not realize that the original program had used `complex128`s, and tried, unsuccessfully, to replicate it with `float64`s.

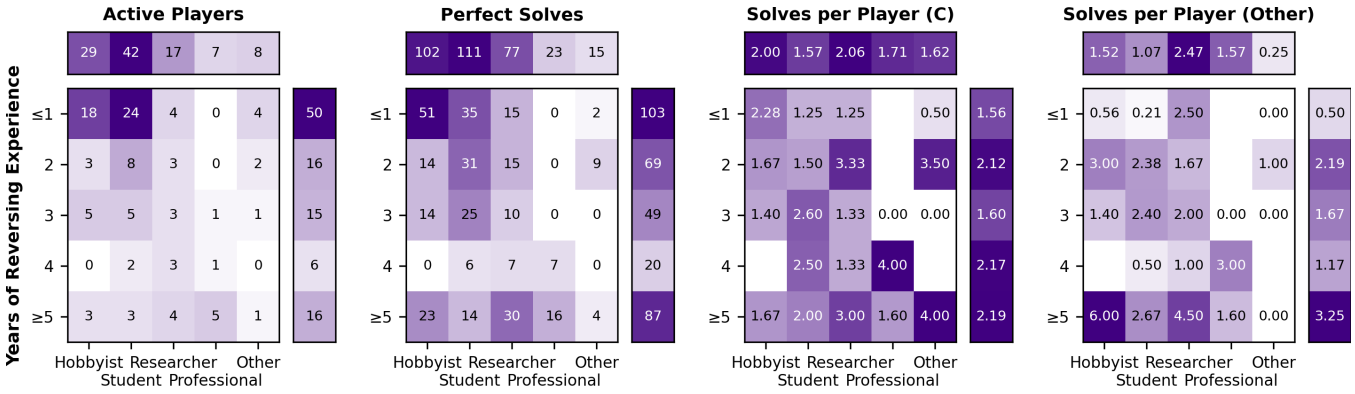

Figure 4: Distribution of active players and perfect submissions, broken down by involvement, experience, and language. "Active" players reached a 20% score or better on at least one challenge. Only one perfect submission is counted per (user, challenge) pair.

the magnitude of the score changes they produced. Figure 5 summarizes the correlation between the two. The big cluster of data points to the left in Figure 5 shows how each submission tended to contain only minor changes from the previous submission, which resulted in an even smaller change in diff score. The average Levenshtein distance was only 47 characters (or 5 lines), and each submission yielded, on average, a 1.5% improvement in diff score over the last.

Score improvement did increase slightly with edit size, but the correlation was very loose, with a Pearson's r of only 0.1. In general, edits were almost as likely to cause the diff score go down as to go up, and small edits were just as capable of causing large score swings as large edits. Adjustments to stack layout are a good example of this effect: changing the size of a single local variable can potentially affect every stack reference in a function.

These results show that reversers tried to approach the solution through a series of small modifications, each testing a hypothesis with a limited scope (e.g., the modification of a single control-flow structure, or the type specification of a single variable).

Function Focus. We were also able to measure which functions reversers focused on over the course of a challenge. Since our disassembler produced per-function output, we were able to calculate per-function diff scores over time, and use these as a proxy for attention: any function seeing frequent changes in diff score was very likely to be the center of a reverser's attention.

The top row of Figure 6 shows the changes in function diff scores over the course of three perfect solutions. The reversers seemed to prefer working on one function at a time—although this may have been encouraged by the per-function view of the DECOMPERSON interface; the (estimated) places where they switched focus are shown on the graphs. This result is confirmed when looking at the number of functions that change between two subsequent submissions, which we summarized in Table 3. Of the 24,265 non-Swift submissions

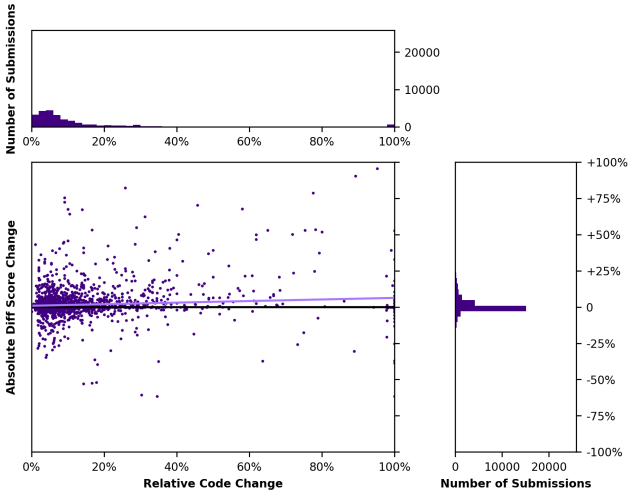

Figure 5: Diff score changes versus code changes, measured as the Jaccard distance on source code lines. For the sake of readability, only 10% of the submissions are visible in the scatter plot. The line of best fit is shown in light purple.

that compile,[9] 19,940 (82%) include changes affecting only a single function. Edits are also highly clustered over time; any function that was modified by one edit has about an 80% chance of being modified again by the next.

More generally, we found that reversers tended to operate in two distinct stages:

- During the first stage, reversers operated at the level of the entire program. In this stage, they would stub out the program, making sure that functions had enough code to successfully compile, and that the functions had the correct type signatures. The latter was particularly important for name-mangled languages: these languages embed the type signature in the function symbol name, and our differ

[9] The library we used to classify code changes did not support Swift.

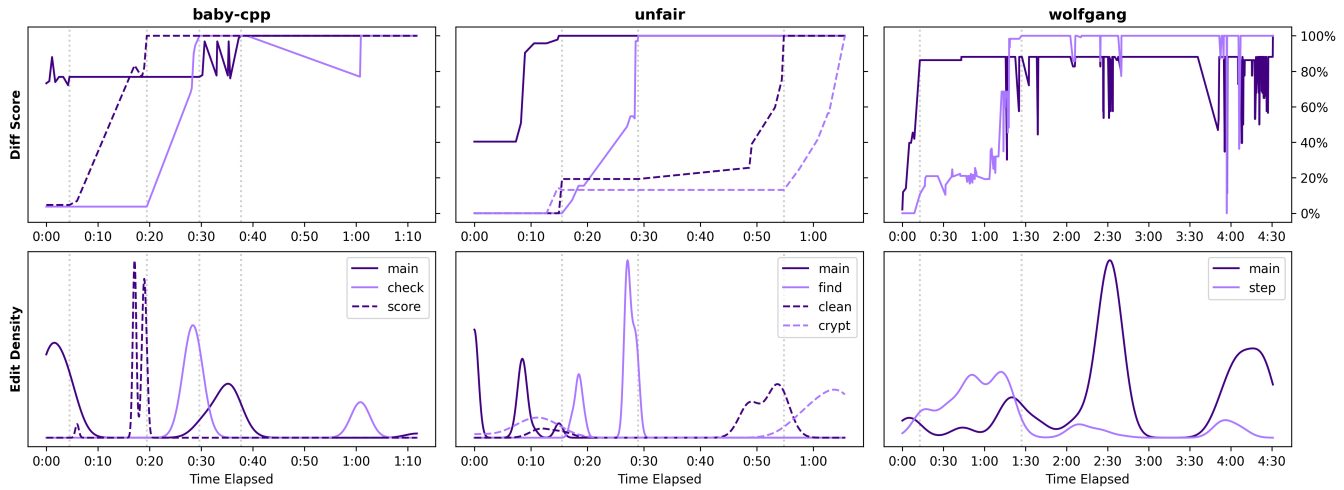

Figure 6: The processes followed by three expert reversers as they solved the `baby-cpp`, `unfair`, and `wolfgang` challenges, respectively. Diff scores by function are shown on top, and edit density per function is shown on the bottom. Submissions that did not compile are omitted. Function focus can be observed both as fluctuations in function diff score and as clusters of edits. Places where the reversers switched focus to a different function are marked with vertical lines.

could not locate functions until the prototype was correct.

During this stage, we would typically see minor increases in the diff scores for all functions. It was also common for the first few submissions to not compile at all—only 37% of the initial submissions did so—and the first submission that did compile would typically contain edits to multiple functions.

- Once the program was stubbed out, reversers would move on to the second stage. In this stage, they would focus on one function until they got stuck or were able to achieve a perfect disassembly match. Once this happened, they would move on to the next function, repeating this step as many times as necessary to fully recreate the program. This pattern is visible in Figure 6, where changes in focus are marked with vertical lines.

These observations mirror the model extracted by Votipka et al. [48]. In their study of the more traditional reverse engineering process, the authors found that reversers operate in three distinct phases: *overview*, *sub-component scanning*, and *focused experimentation*. In the overview phase, reversers acquire a broad overview of the program's functionality, which is then refined in the latter two phases. Our analysis shows that reversers follow a similar approach when decompiling binaries: Our phase one condenses the overview and sub-component scanning stages, as reversers determine what the binary does, and recreate how its functional components fit together. Our phase two corresponds to the focused experimentation stage. Each source code submission is effectively a hypothesis about how a function was implemented, and the response from DECOMPERSON offers evidence either for or against that hypothesis.

	Number of Modified Functions			
Language	0	1	2	≥3
C	540	6,931	443	108
C++	570	4,954	322	219
Go	760	6,200	522	135
Rust	300	1,855	268	138
Swift	—	—	—	—
Total	2,170	19,940	1,555	600

Table 3: The number of edits that touched any given number of functions. The vast majority of edits only modified a single function.

We also observed that the reversers chose functions to focus on based on the binary's control flow graph. Most reversers operated in a breadth-first manner: 75% of the perfect solutions to two-function binaries focused on `main()` first. For binaries with three or more functions, 46% of the solutions focused on `main()` first, while 29% saved it for last. This preference for breadth-first reversing held for all challenges except the C challenge `rootkit`, where depth-first was slightly more popular, and all the Rust challenges, where multi-function edits were more common and function focus was harder to discern. There was no significant difference between the behaviors of beginners and experts; function order seems to be a matter of personal preference.

Edit Classification. The previous analyses highlight the magnitude and location of individual source code changes, but tell us very little about the reversers' intent when making those changes. We would also like to know the semantics of each edit: what specific feature of the binary was the reverser trying to reproduce?

To extract this information, we used Tree-sitter [43], a source code parsing library that supports all of the challenge languages except Swift, which we did not include in this analysis. Given a string of source code, Tree-sitter calculates a "concrete syntax tree," which includes the location of each syntax node within the source string. We could combine this information with the location of an edit to determine which syntax nodes were affected. We then used the types of these nodes—also provided by Tree-sitter—to classify the edit.

We recognize a total of eight semantic categories:

Control Flow: Constructs that redirect the execution of a program, such as conditionals, loops, and exceptions.

Declarations: Definitions of new variables, typically locals. We classify these separately from other statements because they change the layout of a function's stack frame.

Functions: Changes that alter the signature of a function, such as changes to the return type, name, or arguments.

Name Changes: Changes that only affect a single identifier, typically caused by reversers renaming variables.

Statements: Typical imperative programming statements, like assignments, increments, and function calls.

Types: Constructs that define custom data types, or edits to member variables.

Comments: Changes to comments, typically made to temporarily remove code, or to clean up the starter code.

Miscellaneous: Other edits. These are primarily changes to import statements, preprocessor directives, and the like.

A single edit can have multiple classifications, depending on the syntax nodes affected, and can contain multiple instances of the same classification. For example, adding the following C function

```
float hypotenuse(float a, float b) {
    float a2 = a * a
    float b2 = b * b;
    return sqrt(a2 + b2);
}
```

would count as seven edits, namely three function edits (the function and its two parameters), two declarations, one statement (the function call), and one control-flow edit (the return statement).

The ability to classify edits gives us a more detailed view into the reverse engineering process. Figure 7 provides an example. In this case, the reverser had initially focused on the `main()` function, and was able to reach a reasonably high diff score by decompiling its body, as evidenced by the declaration and statement edits. After about ten minutes, however, other functions were needed by `main()`, and the reverser switched to the stubbing phase described previously. This can be identified by the prevalence of function prototype edits, and the fact that they appear in many functions at approximately the same time. The

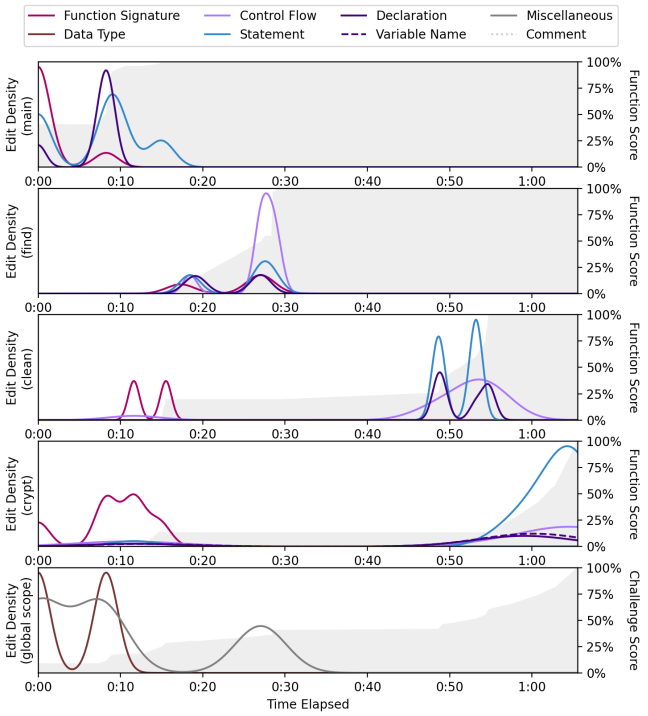

Figure 7: Edit types over time as the top team solves the `unfair` challenge. The diff score of each function (and of the challenge as a whole, in the global row) is shown as a light gray background. The stubbing phase is visible as a spate of function signature edits around the ten minute mark; the reverser then focuses on one function at a time, bringing each one up to 100% with a series of edits to the function body. Another view of this process is shown in the central column of Figure 6.

reverser then went to work on `find()`, then `clean()`, and finally `crypt()`. These latter changes were all function body edits, and classified as variable declarations, control flow changes, and generic statements (assignments and function calls).

In general, participants made most of their function signature edits early in the course of reversing a function, and most of their variable name edits later on, once they had a better idea of what the variables were used for. Data type edits also tended to be made later. Declaration, statement, and control flow edits made up the majority of the reversing work, and remained consistently common, with the reversers showing a slight preference for handling control flow early on and fixing declarations towards the end.

Other patterns of edits appear in specific circumstances. Reversers trying to fix stack layout mismatches, for example, will make lots of edits to variable declarations, while reversers recovering user-defined structs will make mainly data type edits. And some languages have their own idiosyncrasies; during DECOMPETITION, for example, teams fighting with Go's register allocator would often try renaming variables in an attempt to appease it, resulting in a multitude of variable name edits.

Summary. Reversers prefer to make frequent, small code changes, submitting often and refining their mental model of the program based on the resulting assembly changes. Experience is useful, particularly for higher-level languages, allowing reversers to quickly map common assembly patterns to the corresponding source code patterns.

We also found that reversers tend to start work by recreating an outline of the program, providing stubs for all the important functions—a task that could be made easier in a dedicated function stubbing view with a built-in demangler. They then refine these functions one at a time, typically in an order based on the control flow graph.

Within each function, there were two main challenges: to replicate the behavior of the function and to make sure the local variables are in the correct stack slots. These tasks could also be streamlined with dedicated views: a graph-matching algorithm could be used to generate local variable aliases (instead of stack offsets) for use while working on functionality, and variable declarations could likely be reordered automatically to give the correct stack layout. The reversers' concentration on single functions also suggests that humans are able to generate perfect decompilation without relying on the interprocedural context used by automatic decompilers.

7 RQ3: Does Reversing Equal Decompilation?

As mentioned previously, perfect decompilation is a very exacting goal, requiring more effort from the reverser than a traditional reversing task. Since the DECOMPETITION participants were required to generate perfect decompilation for each challenge binary in order to get a perfect score, there is some concern that our observations are not applicable to reversing in general. However, we can show that the process involved is not substantially different.

Traditional reversing tasks, such as binary patching or vulnerability discovery, require extremely precise knowledge of particular "focus points" of a program, namely the points with potential bugs. These points are typically a small fraction of the entire binary, but a successful reverser will need complete knowledge of what the program is supposed to do—as well as how it does it—at each of them. In this light, perfect decompilation is primarily an extension of traditional reverse engineering:

- Reversers already dedicate very close attention to focus points, so asking participants to devote that level of analysis to an entire (small) binary is simply an extension of an existing reversing task rather than a new one.
- We assume that reversers have a code-like internal representation of the behavior of functions, so requiring participants to submit valid source code merely standardizes the language of communication.[10]
- Requiring perfect decompilation makes it trivial to confirm that a participant has completely reversed a function.

The third and final point is the only one where perfect decompilation task differs from traditional reverse engineering. The first two points are extensions of existing tasks, but the third adds a new one: convincing the compiler to generate specific binary output.

Fighting the Compiler. Perfect decompilation represents the highest standard for reverse engineering: a source code that matches the original binary has, by definition, the exact semantics of the original source code. However, it is arguable that such a goal could be too far-fetched, and that equivalent code—measured by unit tests—might be a more realistic goal. The risk here is that reversers aiming for perfect decompilation, such as the DECOMPETITION participants, might spend a significant portion of their time fighting with the compiler, making our observations less representative of the traditional reversing process.

We believed that in general, reversers would follow two strategies: they would either attempt to pass all unit tests first and work on matching assembly later, or they would work directly on achieving a perfect assembly match and thus not pass all test cases until the end. To study this behavior, we selected the sequences of submissions that resulted in a perfect decompilation, and we located the first submission in each sequence that passed all unit tests.

We found that 70% of the reversers earned a 100% test score within the first 90% of their submissions, although some of these may have been assembly-first users who passed tests early by chance. These users spent, on average, the first 51% of their submissions and 60% of their time reaching perfect test scores, and the remainder perfecting their assembly. In contrast, the pool of all reversers took, on average, 69% of their submissions and 70% of their time to pass all unit tests.

The exact ratios varied significantly by language; C++ and Swift had the easiest compilers to mimic (only 27% and 31% of solve time, respectively, was spent on this stage), and Go and Rust had the hardest (45% and 51%, respectively).

Reversing vs. Decompilation. As mentioned in the previous section, when analyzing the submissions, we observed many of the same behaviors as Votipka et al. [48], indicating that we are measuring a similar, if not identical, process.

Furthermore, the DECOMPERSON interface maps nicely onto the traditional reversing process. Reversing studies—all the way back to much earlier program comprehension studies [27]—frame reversing as a series of hypotheses made by the reverser: with DECOMPERSON, each source code submission is a hypothesis about program behavior, and the response from the back-end helps confirm or disprove that hypothesis.

We also have evidence that the diff scores used in DECOMPETITION correlate well with function behavior. After the competition, we wrote unit tests for most of the leaf functions, and

[10] This also prevents reversers from glossing over functionality they don't fully understand.

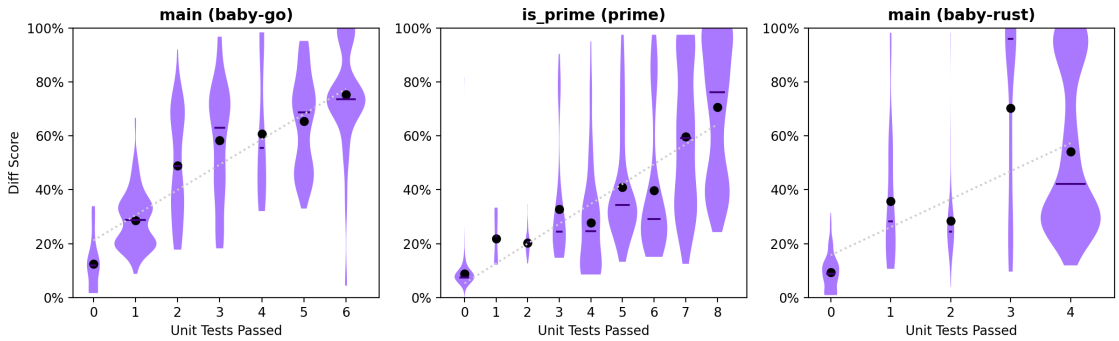

Figure 8: Example distributions of per-function unit test scores and per-function diff scores. Medians are shown as dark purple bars; means are shown as black dots. Lines of best fit are shown in pale gray.

tested the submissions. These test scores were not visible to the participants, and thus could not be used as a decompilation guide, but nevertheless corresponded to the submissions' per-function diff scores; some examples can be seen in Figure 8.

The left two plots show more typical distributions of submission scores. Submissions are balanced across the number of unit tests passed, and the unit test scores are well-correlated with the diff scores, with a Pearson's r of 0.75 and 0.64, respectively. This suggests that in the general case, diff scores, which capture binary similarity, are a good proxy for unit test scores, which capture functionality.

The rightmost plot shows a pathological case. The diff and test scores are still reasonably well-correlated ($r = 0.50$), but the vast majority of submissions passed all the unit tests with a mediocre diff score. This shows that the reversers were able to replicate the behavior of the target function relatively quickly, but then spent a significant amount of time getting `rustc` to generate an exact assembly match.

Summary. Decompilation is a natural extension of traditional reversing. Reversers already generate a code-like representation of the programs they work on, so turning reversing into decompilation is simply a matter of writing this representation in a well-defined language—and recovering a precise representation of an entire function.

Perfect decompilation is no longer purely an extension of the existing process, as it requires more effort to convince the compiler to generate specific output, but we found that even in this case the process correlates well with function comprehension.

8 Threats to Validity

Experimental Setup. While we tried to reproduce standard reversing activities as closely as possible, some differences were unavoidable. For example, running the competition in a web-based editor took people out of their familiar reversing environments. After the competition, some players suggested modifications to the web interface to make it easier to use, while others were interested in integrating DECOMPERSON into their pre-existing reversing environments. However, we believe that the web-based interface did not affect how humans approach the decompilation problem, although it may have influenced the timings recorded during the competition.

Automatic Decompilers. Participants were able to paste the output of automatic decompilers into the DECOMPERSON editor. While this could potentially mean that we were not observing the human reversing process, we found that this was a rare behavior, and that the tools were not always helpful.

As an illustration, Figure 9 shows the maximum similarity between any user's `rootkit` submissions and the output of the Hex-Rays decompiler. Only submissions with over 50% similarity clearly resemble Hex-Rays, and only two users (visible on the right in the first plot) achieved a perfect score with code in this range.

Overall, after analyzing the C submissions, we found that almost 20% of all function decompilation attempts included a submission that clearly resembled tool output. Hex-Rays was the most popular tool, accounting for 10% of all attempts, followed by Ghidra at 7% and Binary Ninja at 3%. However, only 6% of all attempts achieved perfect decompilation with code that resembled tool output;[11] once the output had been edited into perfect decompilation, it was often no closer to the original than a solution that had been written by hand.

Automatic decompilation was not useful for any other languages. Only 2% of C++ function decompilation attempts, for example, attempted to use tool output, and none of these were successful. In fact, there were no perfect submissions for any non-C languages that resembled tool output.

However, traditional reversing tools still proved useful as a reversing aid. Fully 80% of the players who responded to the follow-up survey reported using at least one tool "often" while working on the challenge binaries. Hex-Rays was the most popular tool, used often by about 50% of the respondents; Ghidra and Binary Ninja were used by about 33% each. Most

[11] The vast majority of these successful attempts came from the `prime` and `baby-c` challenges, at 18% and 6% respectively. The other C challenges had an average tool success rate of 1%.

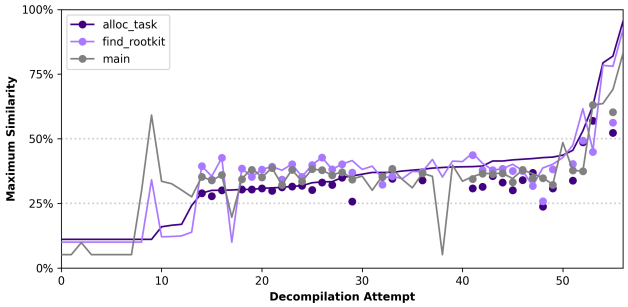

Figure 9: Similarity of all `rooktit` decompilation attempts to Hex-Rays output, measured as Jaccard similarity over character diffs and ordered by similarity of the `alloc_task` function. Maximum similarity of any submission is shown as lines; maximally similar perfect decompilation is shown as points.

respondents used multiple tools, and only two (4%) used none.

Function Symbols. Our custom disassembler uses function symbols to identify which functions to disassemble; therefore, the DECOMPETITION binaries contained function symbols. In practice, these are not always available. Symbols can be "stripped" from real-world binaries—usually to obfuscate them or to reduce their size—and adding meaningful function names is an important step when reversing these.[12]

Having access to function symbols saved our reversers some work. Although the function names used in DECOMPETITION were not particularly descriptive, they still provided some hints about what each function did. Additionally, the function symbols in name-mangled languages encode the types of each function's parameters, which can be quite useful for reversing.

We do not believe that this information significantly altered the decompilation process, but further study, using stripped binaries and a more advanced function identification algorithm, would be needed to definitively show the size of the effect.

Collaboration. DECOMPETITION was an online competition, so we could not prevent users from collaborating on their decompilation. However, we took the following steps to limit this phenomenon: (1) the online interface was not designed to be used collaboratively; (2) players had individual logins; and (3) we used IP addresses to distinguish submissions that were likely done by multiple users.

Overall, more than 80% of all players used the same IP address for the entire competition, and 75% of all decompilation attempts did not include submissions from interleaved IP addresses. The vast majority of the interleaved submissions came from well-established teams, who were accustomed to sharing team accounts in other competitions. We know that the top-scoring team used an informal "lock" protocol in their team chat to prevent conflicting edits to the same challenge; other teams may have used a similar system.

[12] Function symbols are not uncommon, even in a reversing context; dynamically linked binaries contain external function symbols even after stripping, and in 2018, Cozzi et al. found that about 75% of Linux malware was not stripped at all [10].

9 Future Work

DECOMPERSON was developed with DECOMPETITION and humans in mind, but we believe that, with the necessary improvements, it could be used as a general-purpose reversing tool. For example, the current system relies on the user to provide all insights. The DECOMPERSON interface makes it obvious where assembly mismatches occur, but the responsibility of identifying the causes of these mismatches—and of providing code that avoids them—lies solely with the human operator. A more intelligent system would be able to recognize causes of particular classes of mismatches and suggest potential solutions.

Such a system could be trained on the output of human reversers, including the data we collected here. Then, depending on the quality of its suggestions, it could be put to a variety of uses: a more rudimentary system could offer suggestions to novice reverse engineers, helping them until they reach a perfect match; an advanced system could provide likely mutations to power a genetic decompiler, such as BED [38]. This system could be part of a hybrid framework as well, automating decompilation as much as possible before falling back to the user for guidance [39].

This intersection of human and automated decompilation—the same frontier that inspired this paper—still seems the most promising direction for future research: the human insights recorded here can be used to improve future decompilers, which in turn will reduce the workload of their human users, a trend towards combining human input and automated analysis that has recently received substantial attention [14, 39].

10 Conclusions

The process of understanding binary code can be modeled as the process of decompilation. In particular, it is possible to set a precise, measurable goal by requiring that a human perform *perfect decompilation*. We maintain that such a high standard is not unreasonable: our study demonstrated that perfect decompilation of small programs is within reach of human reverse engineers today. We believe that the insights from our experiment (the largest of this kind ever performed) can be used to better understand the human reversing process.

Finally, we offer the dataset we collected during our study to anyone wanting to study the process of human decompilation. We showed some of the data that can be extracted, but we believe that many more insights remain. We hope that it will be used in the spirit of the competition from which it arose: as a small step towards perfect decompilation.

Artifacts

All challenges used in DECOMPETITION, including solutions and test cases, are available here: https://github.com/decompetition/challenges-2020

The environment used to build and test the challenges is available as a Docker container: https://hub.docker.com/r/decompetition/builder-2020

The DECOMPERSON server and web interface can be found here: https://github.com/decompetition/server

The DECOMPERSON disassembler and differ are also available as a standalone tool, which can be found here: https://github.com/decompetition/disassembler

The full (anonymized) submission dataset is also available: https://github.com/decompetition/data

Acknowledgements

The authors would like to thank Noah Spahn and Lukas Dresel for writing and playtesting so many, many challenges; the folks at Vector 35 for giving us space on Binary Ninja Cloud and donating Binary Ninja licenses as extra prizes; and all the random hackers of the internet who for some reason didn't have anything better to do than play our silly reversing game.

This material is based on research sponsored by DARPA under agreement number FA8750-19-C-0003 and by NSF under award CNS-1704253. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] Fiorella Artuso, Giuseppe Antonio Di Luna, Luca Massarelli, and Leonardo Querzoni. Function naming in stripped binaries using neural networks. *CoRR*, 2019.

[2] Pratyay Banerjee, Kuntal Kumar Pal, Fish Wang, and Chitta Baral. Variable name recovery in decompiled binary code using constrained masked language modeling. *arXiv preprint arXiv:2103.12801*, 2021.

[3] Binary Ninja. https://binary.ninja/.

[4] Binary Ninja Intermediate Language. https://docs.binary.ninja/dev/bnil-overview.html.

[5] David Brumley, JongHyup Lee, Edward J Schwartz, and Maverick Woo. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *22nd USENIX Security Symposium (USENIX Security 13)*, 2013.

[6] Adam Bryant. *Understanding How Reverse Engineers Make Sense of Programs from Assembly Language Representations*. PhD thesis, 2012.

[7] Qibin Chen, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Augmenting decompiler output with learned variable names and types. *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[8] Yu Chen, Zhiqiang Shi, Hong Li, Weiwei Zhao, Yiliang Liu, and Yuansong Qiao. Himalia: Recovering compiler optimization levels from binaries by deep learning. In *SAI Intelligent Systems Conference*, 2018.

[9] Cristina Cifuentes. *Reverse compilation techniques*. 1994.

[10] Emanuele Cozzi, Mariano Graziano, Yanick Fratantonio, and Davide Balzarotti. Understanding linux malware. In *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[11] CTFtime. https://ctftime.org/.

[12] EN Dolgova and AV Chernov. Automatic reconstruction of data types in the decompilation problem. *Programming and Computer Software*, 2009.

[13] Javier Escalada, Ted Scully, and Francisco Ortin. Improving type information inferred by decompilers with supervised machine learning, 2021.

[14] Dustin Fraze. Computers and Humans Exploring Software Security (CHESS). https://www.darpa.mil/program/computers-and-humans-exploring-software-security, 2019.

[15] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. Coda: An end-to-end neural program decompiler. *Advances in Neural Information Processing Systems*, 2019.

[16] Cheng Fu, Kunlin Yang, Xinyun Chen, Yuandong Tian, and Jishen Zhao. N-bref : A high-fidelity decompiler exploiting programming structures, 2021.

[17] Ghidra. https://ghidra-sre.org/.

[18] Andrea Gussoni, Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. A comb for decompiled c code. In *15th ACM Asia Conference on Computer and Communications Security*, 2020.

[19] Iman Hosseini and Brendan Dolan-Gavitt. Beyond the c: Retargetable decompilation using neural machine translation. 2022.

[20] IDA Pro. https://www.hex-rays.com/ida-pro/.

[21] Alan Jaffe. Suggesting meaningful variable names for decompiled code: a machine translation approach. In *11th Joint Meeting on Foundations of Software Engineering*, 2017.

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

[23] Deborah S Katz, Jason Ruchti, and Eric Schulte. Using recurrent neural networks for decompilation. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018.

[24] Omer Katz, Yuval Olshaker, Yoav Goldberg, and Eran Yahav. Towards neural decompilation. *arXiv preprint arXiv:1905.08325*, 2019.

[25] Rainer Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. *Journal of Software Maintenance and Evolution: Research and Practice*, 2003.

[26] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Dire: A neural approach to decompiled identifier naming. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

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

[28] Ruigang Liang, Ying Cao, Peiwei Hu, and Kai Chen. Neutron: an attention-based neural decompiler. *Cybersecurity*, 2021.

[29] Alwin Maier, Hugo Gascon, Christian Wressnegger, and Konrad Rieck. Typeminer: Recovering types in binary programs using machine learning. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2019.

[30] Alessandro Mantovani, Simone Aonzo, Yanick Fratantonio, and Davide Balzarotti. Re-mind: a first look inside the mind of a reverse engineer. In *31st USENIX Security Symposium (USENIX Security 22)*, 2021.

[31] Alessandro Mantovani, Luca Compagna, Yan Shoshitaishvili, and Davide Balzarotti. The convergence of source code and binary vulnerability discovery – a case study. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS22)*, 2022.

[32] Alan Mycroft. Type-based decompilation (or program reconstruction via type reconstruction). In *European Symposium on Programming*. Springer, 1999.

[33] Yuhei Otsubo, Akira Otsuka, Mamoru Mimura, Takeshi Sakaki, and Hiroshi Ukegawa. o-glassesx: Compiler provenance recovery with attention mechanism from a short code fragment. 2020.

[34] Davide Pizzolotto and Katsuro Inoue. Identifying compiler and optimization options from binary code using deep learning approaches. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020.

[35] Ashkan Rahimian, Paria Shirani, Saed Alrbaee, Lingyu Wang, and Mourad Debbabi. Bincomp: A stratified approach to compiler provenance attribution. *Digital Investigation*, 2015.

[36] Pemma Reiter, Hui Jun Tay, Westley Weimer, Adam Doupé, Ruoyu Wang, and Stephanie Forrest. Automatically mitigating vulnerabilities in x86 binary programs via partially recompilable decompilation. *arXiv preprint arXiv:2202.12336*, 2022.

[37] Nathan Rosenblum, Barton P Miller, and Xiaojin Zhu. Recovering the toolchain provenance of binary code. In *International Symposium on Software Testing and Analysis*, 2011.

[38] Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, and Alexey Loginov. Evolving exact decompilation. In *Workshop on Binary Analysis Research (BAR)*, 2018.

[39] 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 Conference on Computer and Communications Security (CCS)*, 2017.

[40] Elliot Soloway, Beth Adelson, and Kate Ehrlich. Knowledge and processes in the comprehension of computer programs. *The nature of expertise*, 1988.

[41] M-AD Storey, Kenny Wong, and Hausi A Müller. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 2000.

[42] Claire Taylor and Christian Collberg. Getting revenge: A system for analyzing reverse engineering behavior. In *Malware Conference*, 2019.

[43] Tree-sitter. https://tree-sitter.github.io/.

[44] Freek Verbeek, Pierre Olivier, and Binoy Ravindran. Sound c code decompilation for a subset of x86-64 binaries. In *International Conference on Software Engineering and Formal Methods*. Springer, 2020.

[45] Anneliese von Mayrhauser and A Marie Vans. *Program Unterstanding: A Survey*. Colorado State Univ., 1994.

[46] Anneliese Von Mayrhauser and A Marie Vans. Program comprehension during software maintenance and evolution. *Computer*, 1995.

[47] Anneliese Von Mayrhauser and A. Marie Vans. Identification of dynamic comprehension processes during large scale maintenance. *IEEE Transactions on Software Engineering*, 1996.

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

[49] Tao Wei, Jian Mao, Wei Zou, and Yu Chen. A new algorithm for identifying loops in decompilation. In *International Static Analysis Symposium*, 2007.

[50] Khaled Yakdan, Sebastian Eschweiler, and Elmar Gerhards-Padilla. Recompile: A decompilation framework for static analysis of binaries. In *8th International Conference on Malicious and Unwanted Software:" The Americas"(MALWARE)*, 2013.

[51] 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*, 2015.

A Background Survey

1. What best describes your involvement in computer security?
   (a) Professional
   (b) Researcher
   (c) Student
   (d) Hobbyist
   (e) Other
2. How many years have you been involved in computer security?
   (a) Integer value: 1 or above.[13]
3. How many years have you been involved in reverse engineering?
   (a) Integer value: 1 or above.[13]
4. How much of your paid time is spent reverse engineering?
   (a) I'm not paid to reverse anything.
   (b) A little bit of my workload is reversing.
   (c) Around half of my workload is reversing.
   (d) Most or all of my workload is reversing.
5. How much confidence do you have in your reversing skills?
   (a) I have no idea what I'm doing.
   (b) I'm still a beginner.
   (c) I'm an average reverser.
   (d) I'm better than average.
   (e) I am an expert.
6. How difficult is completely recreating the source code of a small binary?
   (a) Trivial
   (b) Easy
   (c) Moderate
   (d) Difficult
   (e) Impossible
7. How familiar are you with *programming* in the following languages? (Range: 1 (Beginner) to 5 (Expert))
   (a) C
   (b) C++
   (c) Go
   (d) Rust
   (e) Swift
8. How familiar are you with *reversing* the following languages? (Range: 1 (Beginner) to 5 (Expert))
   (a) C
   (b) C++
   (c) Go
   (d) Rust
   (e) Swift
9. How often do you use the following tools when reversing? (Range: 1 (Never); 2 (Sometimes); 3 (Often))
   (a) Angr
   (b) Binary Ninja
   (c) CLI Tools (`objdump`, etc.)
   (d) Ghidra
   (e) Ida / HexRays
   (f) Radare
   (g) Custom Scripts

[13] A programming error prevented users from entering zero.

B Follow-Up Survey

The follow-up survey contained repeats of questions 5–9 from the background survey. Question 9 was re-worded to ask about tools used *during* the competition, and included an item for the DECOMPERSON web interface. There were also free-form text inputs where players could give feedback on the challenges, the interface, or the competition in general.

C Table of Challenges

	Challenges			Non-Blank Lines			Submissions by Diff Score							Language Features
	Name	Functionality	Value	Src.	Asm.	Ratio	All	>0%	>20%	>40%	>60%	>80%	100%	
C	`baby-c`	Integer GCD.	200	24	77	3.21	1,933	1,445	1,357	1,171	943	526	112	arithmetic
	`bitesize`	Overflowable buffer.	100	30	81	2.70	2,803	2,327	2,302	2,056	1,593	760	70	types, buffers
	`prime`	Primality check.	100	20	117	5.85	1,670	1,337	1,307	1,013	432	282	56	arithmetic
	`rootkit`	Linked list search.	100	36	121	3.36	3,654	2,913	2,821	1,560	1,035	585	37	structs
	Subtotal		500	110	396	3.60	10,060	8,022	7,787	5,800	4,003	2,153	275	
C++	`baby-cpp`	Scrabble scoring.	200	49	441	9.00	3,370	2,553	2,259	1,685	1,270	669	41	strings, control flow
	`lambic`	Inventory mgmt.	400	152	647	4.26	815	495	262	158	116	70	4	structs, lambdas, vectors
	`pedigree`	Family tree search.	200	77	337	4.38	1,351	603	511	388	299	183	76	classes, sets
	`streamy`	RPN calculator.	400	146	622	4.26	895	593	467	394	267	142	24	classes, inheritance, streams
	`unfair`	Playfair cipher.	200	67	501	7.48	2,724	1,812	1,343	1,017	715	449	15	strings, tuples
	Subtotal		1,400	491	2,548	5.19	9,155	6,056	4,842	3,642	2,667	1,513	160	
Go	`baby-go`	FizzBuzz.	200	19	122	6.42	1,592	1,073	902	638	456	131	38	arithmetic
	`batsounds`	TCP echo server.	300	44	293	6.66	878	568	460	383	244	149	10	networking, time
	`carshop`	Inventory search.	300	75	400	5.33	726	352	299	191	163	108	8	structs, enums
	`fabulous`	Fibonacci sequence.	200	23	153	6.65	1,322	934	825	612	441	397	15	structs, arithmetic
	`julie`	Julia set ASCII art.	300	79	397	5.03	3,156	2,725	2,481	2,313	2,219	1,718	0	syscalls, complex numbers
	`switcher`	Buggy ROT-13 cipher.	200	32	153	4.78	1,600	1,219	1,110	1,039	937	667	14	strings, switches
	`wolfgang`	Cellular automaton.	300	38	212	5.58	918	717	619	578	446	358	3	strings, arithmetic
	Subtotal		1,800	310	1,730	5.58	10,192	7,588	6,696	5,754	4,906	3,528	88	
Rust	`baby-rust`	Integer parsing.	300	14	222	15.86	1,325	975	765	388	270	237	30	matching
	`habidasher`	Two hash functions.	200	26	292	11.23	498	354	256	167	101	81	7	strings, arithmetic
	`s2ring`	Turing machine.	500	74	796	10.76	1,049	817	672	386	200	133	8	strings, structs, matching
	`toobz`	TCP pipeline.	300	105	385	3.67	652	415	363	335	288	239	8	traits, networking, lambdas
	Subtotal		1,300	219	1,695	7.74	3,524	2,561	2,056	1,276	859	690	53	
Swift	`baby-swift`	Hotdog detector.	300	34	489	14.38	1,221	750	475	360	259	138	14	strings, sets
	`bandate`	Date comparison.	400	55	1,008	18.33	569	353	314	147	85	59	9	structs, dates
	`cardigan`	Luhn checksum.	500	89	1,374	15.44	809	583	485	387	110	37	4	strings, arithmetic
	Subtotal		1,200	178	2,871	16.13	2,599	1,686	1,274	894	454	234	27	
	All Challenges		6,200	1,308	9,240	7.06	35,530	25,913	22,655	17,366	12,889	8,118	603	

Table 4: Descriptions and statistics for the DECOMPETITION challenges.

D Table of Compilers

Language	Compiler	Version	Options
C	`gcc`	9.3.0	`-fno-asm -g`
C++	`g++`	9.3.0	`-fno-asm -g -std=c++17`
Go	`go build`	1.13.8	
Rust	`rustc`	1.43.0	
Swift	`swiftc`	5.2.4	

Table 5: Compilers used during DECOMPETITION. All challenges were built for x86-64 on Ubuntu 20.04. The challenge binaries were compiled with the same options, but all debug information was removed from these before distribution. Note that the `-fno-asm` option only prevents the use of the `asm` keyword in C and C++; we used an additional pre-processing step to prohibit the `__asm` keyword. See the full build scripts in the challenge repository for details: https://github.com/decompetition/challenges-2020