An Observational Investigation of Reverse Engineers' Processes and Mental Models

Daniel Votipka, Seth Rabin, Kristopher Micinski*, Jeffrey S. Foster†, and Michelle L. Mazurek
University of Maryland *Haverford College †Tufts University

KEYWORDS
Reverse Engineering; Program Comprehension.

INTRODUCTION

Reverse engineering is a complex task essential to several software security jobs like vulnerability discovery and malware analysis [14]. While traditional program comprehension tasks (e.g., program maintenance or debugging) have been thoroughly studied [2, 9, 10], reverse engineering diverges from these tasks as reverse engineers do not have access to developers, source code, comments, or internal documentation. Further, reverse engineers often have to overcome countermeasures employed by the developer to make the task harder (e.g., symbol stripping, packing, obfuscation).

Significant research effort has gone into providing program analysis tools to support reverse engineers [1, 3, 5, 11, 12]. However, little work has been done to understand the way they think and analyze programs, potentially leading to the lack of adoption of these tools among practitioners [6, 7].

This paper reports on a first step toward better understanding the reverse engineer's process and mental models and provides directions for improving program analysis tools to better fit their users. We present the initial results of a semi-structured, observational interview study of reverse engineers (n=15). Each interview investigated the questions they asked while probing the program, how they answered these questions, and the decisions made throughout. Our initial observations suggest that

CHI'19 Extended Abstracts, May 4–9, 2019, Glasgow, Scotland UK

ACM ISBN 978-1-4503-5971-9/19/05.
https://doi.org/10.1145/3290607.3313040

- **Hypotheses** - Questions that must be answered or conjectures about what the program does. Generally, reverse engineers might form a hypothesis about the main purpose of some function, or about whether some specific control flow is possible.
- **Simulation methods** - Any process where a participant discusses reading or running the code to determine its function. Simulation is essential to understanding how a program works. Programmers use tools like debuggers and strategies like line-by-line manual investigation to understand how a program works. Reverse engineers apply similar strategies, and we seek to identify these.
- **Beacons** - Patterns or tells that a reverse engineer recognizes, allowing them to quickly understand the functionality of some code without stepping through it line-by-line.
- **Decision points** - Moments where the participant must decide between one or more actions during their process. This can include deciding whether to delve deeper into a specific function or which simulation method to apply to validate a new hypothesis.
- **Resources** - Any documentation or information source external to the code that the reverse engineer uses to answer a question about the program's functionality.

Sidebar 1: Items of interest

reverse engineers rely on a variety of reference points in both the program text and structure as well as its dynamic behavior to build hypotheses about the program's function and identify points of interest for future exploration. In most cases, our reverse engineers used a mix of static and dynamic analysis—mostly manually—to verify these hypotheses. Throughout, they rely on intuition built up over past experience. From these observations, we provide recommendations for user interface and program analysis improvements to support the reverse engineer.

METHODOLOGY

We employ a semi-structured interview protocol, designed to gain detailed insights into reverse engineering experts' processes. We conducted these interviews between October 2018 and January 2019. In the interest of a rigorous qualitative process, we interviewed reverse engineers until we stopped seeing new themes emerge [4, pg. 113-115].

Recruitment

We recruited participants through online forums, associated organizations, and relevant conferences. Respondents were first asked to complete a short questionnaire, which included questions about their level of expertise, years of experience, and demographics. We selected from survey respondents at random. However, we weighted our selection to favor participants with more skill and years of experience to ensure we primarily interviewed experts.

Interview Protocol

We implemented a modified version of the Critical Decision Method, which is intended to reveal expert knowledge by inquiring about specific cases of interest [8]. We asked participants to choose an interesting program they recently reverse engineered, and had them demonstrate the steps they took while we observed. As they walked us through the program, we asked a number of directed questions to probe their mental models. Interviews lasted one hour and 15 minutes on average.

We conducted two pilot interviews prior to the full study. After each pilot, we made adjustments so as to use appropriate terminology and improve question flow. No changes to the protocol were necessary after the second interview. For this reason, we included the second protocol interview in our main study data.

Program background. We began by asking participants to describe the program that they chose to reverse engineer. This included questions about the program's functionality, the size of the program, what tools (if any) they used, and whether they worked with others.

Reverse engineering process. We then asked them to walk us through their process. To allow us to observe behaviors that might be subconscious, interviewees shared their screen. We encouraged

PARTICIPANTS

Table 1 shows our 15 participants' demographics. Table 2 lists the set of programs described by each participant.

ID[1]	Gender: Age:Race[2]	Educ.[3]	Exp Skill	Exp (yrs)	Src.[4]
P1	M:18-29:W	B.S.	4	7	C
P2	M:18-29:W	H.S.	4	8	C
P3	F:18-29:W	M.S.	4	6	C
P4	M:30-39:W	B.S.	5	11	C
P5	M:18-29:W	M.S.	5	6	F
P6	M:18-29:B	SC	4	10	F
P7	M:18-29:A	M.S.	5	10	F
P8	M:18-29:W	Assoc.	4	5	F
P9	M:18-29:W	B.S.	4	14	F
P10	M:30-39:W	M.S.	5	15	O
P11	M:40-49:W	Ph.D.	3	10	F
P12	M:30-39:W	B.S.	3	8	F
P13	M:30-39:H	B.S.	5	21	F
P14	M:18-29:W	M.S.	4	5	F
P15	M:18-29:A	H.S.	3	4	F

[1] IDs are coded in date order
[2] W: White, B: Black, A: Asian, H: Hispanic
[3] H.S.: High School, Assoc.: Associate's, SC: Some College, B.S.: Bachelor's, M.S.: Master's
[4] Recruitment method – O: Related Organization, F: Public Reverse Engineering Forum, C: Related Conference

Table 1: Participant demographics.

participants to share their entire process even if a particular speculative step did not eventually support their final goal.

Items of interest. We chose to focus on the items given in Sidebar 1 which are important to the reverse engineering process. For each item observed, we asked participants to explain the reasoning behind any conclusions they reached or actions taken.

Data Analysis

We plan to complete a systematic analysis of our interview transcripts using an iterative open coding method to identify themes [13, pg. 101-122]. However, this analysis is not yet complete. The initial observations outlined in the next section are simply high-level themes that stood out to both the interviewer and a second researcher who reviewed all interview recordings. Therefore, our observations represent a lower bound of theme prevalence.

INITIAL OBSERVATIONS

An initial review of themes from our interviews highlighted the fact that reverse engineers perform an iterative process: They begin with questions and related hypotheses, evaluate them by simulating the program, and recognize information through beacons, which inform new questions and hypotheses and require decisions to be made regarding which paths to investigate further. This loop is depicted in Figure 1.

Questions/Hypotheses

Reverse engineering is inherently a process of knowledge discovery; reverse engineers begin with high-level goals and questions which lead to more specific questions, whose answers feed back to their higher-level agenda.

Began with general questions. At the start, many reverse engineers begin with broad questions to understand the program's functionality. Most of our participants started by looking at the API calls, variable names, and strings included (N=8).

Other reverse engineers first want to know what the program does under normal conditions, asking "What happens when you run the program?"(N=4) or "What does the main function do?"(N=4). These questions provide a starting point for familiarization and allow identification of more specific lines of inquiry.

As they learned more, they asked more specific questions. As reverse engineers learn about the way the program functions, they identify points of interest and ask specific questions about the program state at those points. For example: "What are the possible values of variable X at point Y?"(N=6), "Can

ID	Program[1]	Reason[2]	Size
P1	R	I	3.4MB
P2	S	E	143MB
P3	C	F	95KB
P4	C	E	13KB
P5	A	E	11MB
P6	C	E	1MB
P7	S	F	46MB
P8	E	E	15MB
P9	E	E	9.6MB
P10	R	I	100KB
P11	B	F	27KB
P12	S	F	693KB
P13	A	E	50MB
P14	R	I	234KB
P15	M	F	5MB

[1] R - Ransomware, S - system-level binary, C - Challenge binary, A - Desktop application, E - Embedded device firmware, M - Mobile application, B - Botnet client

[2] I - Indicators of compromise, E - Exploitation, U - Fully understand functionality

Table 2: Programs discussed by participants

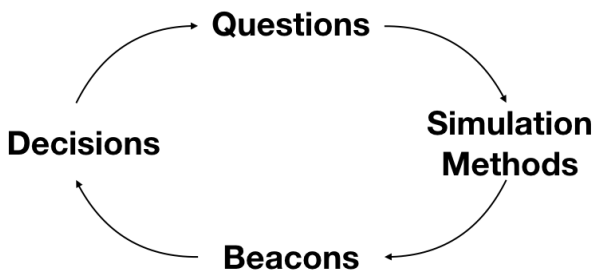

Figure 1: Illustration of the loop reverse engineers iterate through when analyzing programs

user data get to point X?"(N=5), or "Is a given control flow path possible?"(N=3). P2 asked all three questions after identifying a potentially vulnerable function. To determine whether an exploit was possible, he had to know if the function was ever called, if user input was passed to it, and how he could manipulate that input.

Simulation Methods

Once they determine a question to answer, the next step is to simulate—or in some cases actually run—the program.

Most scan for "interesting" components. Many reverse engineers look over the code for named components (function calls, variables, strings), constants, and control flow structures, without focusing on how they were used (N=12). Some also execute the program to determine the set and context of strings displayed and APIs called (N=4). This is used as an initial pass to identify "interesting" components for more detailed review.

Some only mentally execute instructions related to specific data flows. Reverse engineers rarely read the program line-by-line, and they only do it for very small, complex sub-components (N=2). Instead, when they read the code, they focus on specific data flows; skipping any line that does not influence the data of interest (N=8).

Some switched between static and dynamic analysis. Reverse engineers regularly switch between static and dynamic analysis with the results of one feeding the other (N=5). Two participants (P8 and P14) ran the debugger side-by-side with program code so they could quickly switch between the two; scanning the code for interesting points and using the debugger to look up concrete values.

Beacons

As reverse engineers simulate the program, they recognize a variety of beacons that allow them to make additional inferences.

Similar beacons to prior program comprehension literature. As was the case with developers in prior work, reverse engineers utilize named values (strings, variables, and APIs) (N=10), unique constants (N=3), and patterns of variable manipulations (N=2). To give an example of what these beacons look like, Figure 2 shows a simplified version of an encryption routine. The named variables key and cipherText indicate that this is likely an encryption function. The fact that it performs XORs on individual characters tells the reverse engineer that it is an XOR-based cipher.

Considered additional beacons. In addition to the beacons identified previously, reverse engineers also make inferences based on control flow structures (N=3), patterns in concrete values in the execution

```
key = userinput()
cipherText = ""
zipped = zip(*map(bytearray, [data, key]))
for a,b in zipped:
        cipherText += chr(a^b)

print cipherText
```

Figure 2: Sample code for XOR cipher to highlight beacons recognized by reverse engineers

- Reverse engineers utilize a wide variety of beacons to make inferences about program function.
- Reverse engineers combine static and dynamic analyses, using the results of one to feed the other.

Sidebar 2: Highlighted Results

trace (N=5), and the code's position within the program flow (N=2). Returning to Figure 2, if the code is part of program initialization, then it might decrypt stored values or create a session key. If the code is in the main loop, then it is probably for encrypted communication or logging. This was the reasoning used by P10 to differentiate encryption routines for initial server authentication as opposed to file encryption (the main task of the ransomware). These additional beacons are likely used because information is more scarce than in a traditional development setting.

Decisions

From their additional inferences, reverse engineers consider further questions and hypotheses to test, but have to make decisions about how to most effectively use their time.

Made based on prior experience. In most cases, reverse engineers decide routes of investigation based on prior experience of which are most "interesting" or likely to lead to their end goal. For example, they might step into a function if it has "interesting" inputs (N=5), is surrounded by relevant named values (N=2), calls a function of interest (N=3), or has output that is a condition of a control flow path of interest.

Rely on heuristics to bootstrap. When reverse engineers do not notice any similarities with prior work, they make decisions based on heuristics, such as function size (N=1) or complexity (N=2).

RECOMMENDATIONS

Two significant results, given in Sidebar 2, stand out from our findings. These results provide valuable insights for both developers of reverse engineering user interfaces and program analyses.

Make beacons easier to find and process. As beacon processing plays a significant role in reverse engineering, UIs should provide listing and searching functions for all beacons—not just strings and APIs. Additionally, program analyses that highlight similarities between code structures and automatically infer beacon semantic information should be prioritized.

Support integration of static and dynamic analyses. Because reverse engineers regularly switch between static and dynamic methods, their UIs should add support for interactions that allow seamless switching, automatically passing context information. Similarly, program analysis developers should create abstractions that allow both static and dynamic traces to be processed together.

ACKNOWLEDGEMENTS

We thank Kelsey Fulton and the anonymous reviewers for their helpful feedback. This research was supported in part by a UMIACS contract under the partnership between the University of Maryland and DoD.

REFERENCES

[1] Nuno Antunes and Marco Vieira. 2009. Comparing the Effectiveness of Penetration Testing and Static Code Analysis on the Detection of SQL Injection Vulnerabilities in Web Services. In *Proc. of the 15th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC '09)*. IEEE Computer Society, 301–306. https://doi.org/10.1109/PRDC.2009.54

[2] Vairam Arunachalam and William Sasso. 1996. Cognitive Processes in Program Comprehension: An Empirical Analysis in the Context of Software Reengineering. *Journal on System Software* 34, 3 (Sept. 1996), 177–189. https://doi.org/10.1016/0164-1212(95)00074-7

[3] Andrew Austin and Laurie Williams. 2011. One Technique is Not Enough: A Comparison of Vulnerability Discovery Techniques. In *Proc. of the 5th International Symposium on Empirical Software Engineering and Measurement (ESEM '11)*. IEEE Computer Society, 97–106. https://doi.org/10.1109/ESEM.2011.18

[4] Kathy Charmaz. 2006. *Constructing Grounded Theory: A Practical Guide Through Qualitative Analysis*. SagePublication Ltd, London. 113–115 pages.

[5] Adam Doupé, Marco Cova, and Giovanni Vigna. 2010. Why Johnny Can'T Pentest: An Analysis of Black-box Web Vulnerability Scanners. In *Proc. of the 7th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'10)*. Springer-Verlag, 111–131.

[6] Ming Fang and Munawar Hafiz. 2014. Discovering Buffer Overflow Vulnerabilities in the Wild: An Empirical Study. In *Proc. of the 8th International Symposium on Empirical Software Engineering and Measurement (ESEM '14)*. ACM, Article 23, 10 pages. https://doi.org/10.1145/2652524.2652533

[7] Munawar Hafiz and Ming Fang. 2016. Game of detections: how are security vulnerabilities discovered in the wild? *Empirical Software Engineering* 21, 5 (2016), 1920–1959. https://doi.org/10.1007/s10664-015-9403-7

[8] Gary A Klein, Roberta Calderwood, and Donald Macgregor. 1989. Critical decision method for eliciting knowledge. *IEEE Transactions on Systems, Man, and Cybernetics* 19, 3 (1989), 462–472.

[9] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information During Software Maintenance Tasks. *IEEE Transactions on Software Engineering* 32, 12 (Dec. 2006), 971–987. https://doi.org/10.1109/TSE.2006.116

[10] Thomas D. LaToza, David Garlan, James D. Herbsleb, and Brad A. Myers. 2007. Program Comprehension As Fact Finding. In *Proc. of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)*. ACM, New York, NY, USA, 361–370. https://doi.org/10.1145/1287624.1287675

[11] Gary McGraw and John Steven. 2011. Software [In]security: Comparing Apples, Oranges, and Aardvarks (or, All Static Analysis Tools Are Not Created Equal. http://www.informit.com/articles/article.aspx?p=1680863 (Accessed 02-26-2017).

[12] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. 2004. A Comparison of Bug Finding Tools for Java. In *Proc. of the 15th International Symposium on Software Reliability Engineering (ISSRE '04)*. IEEE Computer Society, 245–256. https://doi.org/10.1109/ISSRE.2004.1

[13] Anselm Strauss and Juliet Corbin. 1998. *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Vol. 15. Newbury Park, CA: Sage. 101–142 pages.

[14] Daniel Votipka, Rock Stevens, Elissa M. Redmiles, Jeremy Hu, and Michelle L. Mazurek. 2018. Hackers vs. Testers: A Comparison of Software Vulnerability Discovery Processes. In *Proc. of the 39th IEEE Symposium on Security and Privacy (SP '18)*. 374–391. https://doi.org/10.1109/SP.2018.00003