

REMOTELY OBSERVING REVERSE ENGINEERS TO
EVALUATE SOFTWARE PROTECTION

by
Claire Taylor

A Dissertation Submitted to the Faculty of the
DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfillment of the Requirements

For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

2 0 2 2

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Dissertation Committee, we certify that we have read the dissertation prepared by: Claire Taylor, titled: *Remotely Observing Reverse Engineers to Evaluate Software Protection*

and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

Christian Collberg

Christian Collberg

Kate Isaacs

Kate Isaacs

Saumya Debray

Debray

Carlos Scheidegger

May 27, 2022

Date: _____

May 16, 2022

Date: _____

May 16, 2022

Date: _____

May 19, 2022

Date: _____

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Christian Collberg

Christian Collberg

Dissertation Committee Chair

Computer Science

May 27, 2022

Date: _____

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED:

A handwritten signature consisting of two stylized, cursive initials, possibly 'C' and 'J', written over a single horizontal line.

ACKNOWLEDGMENTS

In no particular order, I would like to thank my faculty advisor, Christian Collberg; the other members of my Graduate Committee, Kate Isaacs, Carlos Scheidegger, and Saumya Debray; and my family, including my parents, siblings, and those close to me though unrelated by blood; Lawrence Livermore National Laboratory, which has supported me throughout my time working on graduate degrees, and the University of Arizona Department of Computer Science — particularly Richard Snodgrass, with whom I worked on the ANTARES project — and College of Law — particularly Derek Bambauer, with whom I had the pleasure of founding the Journal of Emerging Technology; and Robert Glennon, with whom I had the pleasure of working with on water law.

Christian and the rest of my Committee provided crucial guidance throughout my years at the University in addition to providing support with research and teaching associateships. Christian introduced me to an area of research where I could apply my set of skills and expertise in order to make a large impact with this resulting research. He also spent many late nights helping me edit not just this dissertation, but also the other papers and work we generated along the way. I look forward to future collaborations with him — and with the rest of my Committee as well who recognized the potential for the methods this dissertation establishes.

My parents — John and Suzanne Taylor — have always been incredibly supportive in countless ways, and without that support I could not have made it through these years of graduate school. Additionally thanks go to Amanda Tice, Austin Herry, Khalilah Taylor, Morgaine Delo, and Vee Burke, who are all close enough to be considered family and who all lent their support whenever I needed it most, particularly on long days of deadline-driven work.

Finally, I would be remiss to not mention the Fierce Pride soccer team (sponsored by the Southern Arizona Gender Alliance) and somewhat associated Wednesday night soccer group. Both of these gave me reason to take a break from long hours programming, writing, and editing in front of way too much blue screen light and stretch. I dedicate this dissertation in part to Santo, a member of this group who passed in March of this year. Nobody else quite kept up with me like you did, and you are missed.

CONTENTS

LIST OF FIGURES	10
LIST OF TABLES	18
ABSTRACT	20
CHAPTER 1. INTRODUCTION	22
1.1. Experimental Methodology	24
1.2. Research Questions	26
1.3. Current Results	31
1.4. Dissertation Overview	35
CHAPTER 2. BACKGROUND AND RELATED WORK	36
2.1. Software Protection Techniques	37
2.1.1. Code Obfuscation	38
2.1.2. Code Diversification	40
2.1.3. Tamper Proofing	40
2.1.4. Whiteboxing	41
2.2. Code Protection Tools	41
2.2.1. Tigress	42
2.2.2. ASPIRE	47
2.2.3. Irdeto	48
2.3. Reverse Engineering Techniques	48
2.3.1. Static Analysis	49
2.3.2. Dynamic Analysis	50
2.3.3. Symbolic Execution	51
2.3.4. Concolic Execution	52
2.4. Evaluating Obfuscation Strength	52
2.4.1. Provable Security	54
2.4.2. Attack-Defense Scenarios	59
2.4.3. Software Metrics	60
2.4.4. Human Studies	61
2.5. Cyber Attack Models	67
2.5.1. Attack Trees	68
2.5.2. Attack Graphs	69

CONTENTS—Continued

2.5.3. Petri Nets	70
2.6. Building Models from Human Behavior Observation	73
2.6.1. Observation Methods	73
2.6.2. Analysis Methods	78
2.7. Summary	85
CHAPTER 3. THE CATALYST DATA COLLECTION ENGINE	86
3.1. Requirements, Goals, and Assumptions	86
3.2. Architecture	89
3.3. The Endpoint Monitor	92
3.3.1. Sourcing the Data in Heterogenous Computing Environments	93
3.3.2. Storing the Data	94
3.3.3. Building Installers and Compatibility Issues	95
3.3.4. Performance Evaluation	96
3.3.5. Endpoint Monitor Performance Measured in Deployment	98
3.4. Data Uploading Method and Performance	106
3.4.1. Implemented Solution	108
3.4.2. Streaming Data to the Server	110
3.5. Summary	116
CHAPTER 4. REVERSE ENGINEERING CHALLENGES	117
4.1. Study Acknowledgements	119
4.2. Challenge Types	119
4.3. Submission Evaluation	120
4.3.1. Functional Equivalence	121
4.3.2. Code Equivalence	125
4.4. Announcement and Advertisement Methods	130
4.5. <i>RevEngE Pilot</i> Study	130
4.5.1. <i>RevEngE Pilot</i> Results	133
4.6. <i>RevEngE CTF</i>	134
4.6.1. <i>RevEngE CTF</i> Results	135
4.7. The <i>Grand Re</i> Challenge	136
4.7.1. The <i>Grand Re</i> Results	137
4.8. Data Collected from the Events	142
4.9. Ghent University Student Study	147
4.10. Summary	148

CONTENTS—Continued

CHAPTER 5. TOOLS FOR VISUALIZATION AND ANALYSIS	149
5.1. Visualization and Analysis Goals	150
5.2. Loading and Environment Considerations	154
5.3. Visualization User Interface	156
5.4. Petri Net Generation and View	160
5.4.1. Annotations and Petri Nets	161
5.4.2. Annotation Relationship Types	161
5.4.3. Generating Petri Nets from Annotations	165
5.4.4. Petri Net Generation Algorithm and Performance	166
5.4.5. Petri Net Aggregation and Simplification	181
5.4.6. Petri Net Visualization	183
5.4.7. Nesting Considerations	186
5.5. Summary	188
CHAPTER 6. ANALYZING CHALLENGE DATA	191
6.1. Current Results	191
6.1.1. <i>RevEngE CTF</i> Deobfuscation Challenge	192
6.1.2. <i>Grand Re</i> Sessions	193
6.2. Ambiguity in Annotation	197
6.2.1. Grounded Theory Coding	198
6.2.2. Annotation Guidelines	199
6.2.3. Automating Annotations	200
6.3. Summary	200
CHAPTER 7. PERFORMANCE ANALYSIS	202
7.1. Monitoring Facilities	202
7.2. Experimental Setup	205
7.3. Performance Results	205
7.3.1. Endpoint Monitor CPU and Memory Usage	205
7.3.2. Endpoint Monitor Thread Usage	209
7.3.3. Endpoint Monitor Page Faults	213
7.3.4. Achievable Write Rates	218
7.4. Persistent Storage Performance	221
7.5. Summary	224

CONTENTS—*Continued*

CHAPTER 8. FUTURE WORK	225
8.1. Speeding up Annotation	225
8.1.1. Preprocessing Screenshots	226
8.1.2. Crowdsourcing Annotations	228
8.1.3. Machine Learning and Semi-supervised Annotation	228
8.2. Validating Generated Models	229
8.3. Evaluating Visualizations	230
CHAPTER 9. CONCLUSION	232
9.1. Research Question 1	232
9.2. Research Question 2	233
9.3. Research Question 3	234
9.4. Research Question 4	235
9.5. Research Question 5	236
9.6. Released Data	238
APPENDIX A. CATALYST DATA DICTIONARY	239
APPENDIX B. CURRENT IRB APPROVAL	241
APPENDIX C. STUDENT CONSENT DOCUMENT	243
APPENDIX D. GRAND RE CHALLENGE DATA SUMMARY	245
APPENDIX E. CATALYST ER DIAGRAMS	250
APPENDIX F. TIGRESS TRANSFORMS	253
APPENDIX G. REVENGE STUDY CHALLENGE GENERATION SCRIPT 1: EASY PASSWORD EXTRACTION	254
APPENDIX H. REVENGE STUDY CHALLENGE GENERATION SCRIPT 2: MEDIUM PASSWORD EXTRACTION	256
APPENDIX I. REVENGE STUDY CHALLENGE GENERATION SCRIPT 3: DE- OBFUSCATION AND DECOMPILATION	259

CONTENTS—*Continued*

APPENDIX J. REVENCE CTF STUDY DECOMPILE CHALLENGE CONTROL FLOW GRAPHS	261
APPENDIX K. GHENT UNIVERSITY REVERSE ENGINEERING ASSIGNMENT	264
APPENDIX L. GHENT UNIVERSITY ASSIGNMENT RESULTS	268
BIBLIOGRAPHY	271

LIST OF FIGURES

FIGURE 1.1. The big picture: Researchers generate challenges which experts solve while generating traces for analysis.	25
FIGURE 2.1. A simple C program.	42
FIGURE 2.2. An example set of tigress arguments which obfuscate a given file with encoded arithmetic, virtualization, and self-modifying code.	43
FIGURE 2.3. The simple program shown in Figure 2.1 after obfuscation with the tigress script in Figure 2.2.	44
FIGURE 2.4. The control flow graph here derives from a simple, sample program with a single loop, a few arithmetic operations, and a print function at the end. LLVM [200] compiled this program and generated the static control flow graph.	45
FIGURE 2.5. This control flow graph resulted from obfuscating the program in Figure 2.4 with a virtualization transformation. Virtualization — a type of instruction transformation — transforms blocks of code into virtual instructions; a dispatcher (the most connected block near the top in the center) delegates control to these virtual instructions. During execution, the virtual instructions executed functionally mirror the non-obfuscated execution; ideally, this virtualized control flow is more difficult for reverse engineers to analyze.	46
FIGURE 2.6. A basic, single action Petri net in its initial state and after an actor “fires” its action, moving the token for the actor from the input to the output state. Transitions are shown in purple, transitions in red, and tokens in yellow.	71
FIGURE 2.7. An example Petri net proposed (and reproduced from) prior work on RevEngE. [307] This Petri net does not reflect real-world observations, but instead serves as an example and output goal for reverse engineering studies.	72
FIGURE 2.8. Individuals use methods to complete tasks on their devices. The research presented here attempts to capture the “Methods” used in reverse engineering.	74
FIGURE 2.9. Individuals using devices generate time-series data as traces. Analysis methods focus on extracting tasks from these traces and then integrating tasks and raw data into models.	79

LIST OF FIGURES—Continued

FIGURE 3.1. Catalyst collects several types of data while subjects use their devices. A more precise explanation of the data sources and fields collected for them will be presented in 3.3 and 9.6 respectively. The exact structure of this data in the database, including implementation details omitted here, can be viewed in Appendix E’s entity-relationship (ER) diagrams. For later consideration in visualization and analysis algorithms, D_{total} is the set of sets of datapoints — it contains the sorted lists of all datapoints from all data types.	90
FIGURE 3.2. The main components of Catalyst include the Endpoint Monitor in purple, the Back End in black, and the Front End in red. Humans are colored yellow, local data flows green, and web data flows blue. Note that currently Catalyst uses local databases for persistence, but this could be reconfigured as remote. These implement the fundamental of Catalyst — capturing HDI data (of the type illustrated in 3.1) and presenting it to analysts in useful ways which often involve annotating that data.	91
FIGURE 3.3. To use Catalyst, users sign up via a web application, download and run an installer, and run the endpoint, which continually uploads data that it collects from users’ devices. The red arrows indicate requests/messages and responses using standard protocols such as the hypertext transfer protocol (HTTP) or its websocket (WS) extension, while blue arrows indicate processes running on client or server devices.	92
FIGURE 3.4. The Endpoint Monitor’s CPU use over time from participants in The Grand Re Challenge (see 4.7) colored by environment type with averages listed.	100
FIGURE 3.5. A zoomed version of Figure 3.4 which crops most of the startup overhead data points in order to better view patterns in the data.	101
FIGURE 3.6. The Endpoint Monitor’s memory use during the Grand Re.	102
FIGURE 3.7. The CPU use for MySql, a dependency used by the Catalyst Endpoint Monitor, from participants in The Grand Re Challenge, colored by environment type with averages listed at the bottom.	103
FIGURE 3.8. MySql’s memory use during The Grand Re.	104
FIGURE 3.9. This histogram shows the average screenshot insertion and storage latencies recorded on subjects’ devices during The Grand Re. The bins are logarithmic scale; while most latencies show less than two seconds, a significant number of operations took more than that.	105

LIST OF FIGURES—Continued

FIGURE 3.10. An illustration of the upload algorithm showing datapoints arranged on a timeline ($d_1, d_2 \dots d_x$) and attempted uploads depicted by red lines marking upload time periods. When a chunk of data fails to upload, the chunk time interval is halved, resulting in chunk sizes roughly half of the initial size. When the chunk succeeds, the chunking algorithm doubles the time interval.	114
FIGURE 3.11. The Endpoint Monitor chunks data based on timestamps. Initially, the Endpoint Monitor sends the chunk between $Send_0$ and $Send_1$. Then, it sends the chunk between $Send_1$ and $Send_2$. Datapoints all have insertion timestamps. ACID compliance is assumed here, preventing d_{x+1} from being inserted after a query terminating at $Send_1$, where $Send_1 < Timestamp_{current}$	115
FIGURE 4.1. The RevEngE system generated challenges automatically from Tigress configuration scripts, served the resulting obfuscated code, and graded results that subjects submitted. Diagram reproduced from original in [307].	131
FIGURE 4.2. The home page for The <i>Grand Re</i> Engineering Challenge.	137
FIGURE 4.3. Challenge descriptions for The <i>Grand Re</i> round 1.	138
FIGURE 4.4. Challenge descriptions for The <i>Grand Re</i> round 2.	139
FIGURE 4.5. A histogram of the number of process attribute data points uploaded by all users for The <i>Grand Re</i> per day. Process data is sampled on a poll and shows user device activity.	141
FIGURE 5.1. The overall visualization and analysis process: (1) Select sessions to view on the session selection table; (2) visualize those sessions; (3) annotate the sessions in the visualization; (4) generate Petri nets from those annotations.	151
FIGURE 5.2. The overall visualization with two sessions loaded in the default timeline view, consisting of (A) the timelines, (B) the timeline legend, (C) screenshot preview, (D) all-sessions summary data, (E) data highlights, and (F) the options. Not in the screenshot are the process graph, data details table, animation view, and Petri net visualization.	153
FIGURE 5.3. The visualization begins with loading and showing small-sized index data before using multiple, asynchronous, throttled requests to load and store the rest of the data. Finally, when all of the data has been loaded, the entire dataset for the selected sessions is visualized.	155

LIST OF FIGURES—*Continued*

FIGURE 5.4. The timeline view presents each sessions' windows, screenshots, and annotations on a time scale axis.	156
FIGURE 5.5. The process graph right after loading and showing time selected and mouseover tooltip. Green circles indicate a process's UI becoming the active window, and red circles indicate it exiting the active window status.	157
FIGURE 5.6. The same process graph in 5.5 showing a time selection on the timeline and a mouse over a process data point.	158
FIGURE 5.7. The animation view: (A) plays a video of screenshots with mouse clicks superimposed; (B) contains the timeline with clickable seeking and brushable time selection for annotation entry, (C) contains text input; (D) has the top 5 CPU use processes; and (E) contains play/pause buttons, a toggle for the task/annotation creation menu, and active annotations and window information.	159
FIGURE 5.8. An annotation, as pertains to the algorithms described, consists of a start and end node on a timeline.	160
FIGURE 5.9. Given an annotation with a {Task, Goal} tuple, a corresponding Petri net consists of a Transition and a State; the Task, describing the action, becomes the Transition, while the Goal, being the outcome, is added to the State.	162
FIGURE 5.10. A task with a start and end point lying before and after another has a demonstrates a hierarchical relationship, where the longer task is a parent and the shorter a child.	163
FIGURE 5.11. Tasks with either a start or end point — but not both — in between the start and end points of another task are considered concurrent to the other task.	163
FIGURE 5.12. Tasks where both the start and end points come before or after the start and end points of another task are considered sequential.	164
FIGURE 5.13. Task 2 is required to start and finish Task 3; the steps to complete Task 1 are Task 2 followed by Task 3.	166
FIGURE 5.14. Same as Figure 5.13, except Task 4 is also required to complete Task 1 but has no other relationship to Tasks 2 or 3; it is being done concurrently and independently.	167
FIGURE 5.15. Both Task 2 and Task 4 are predecessors of Task 3, but are independent of each other due to concurrency.	168

LIST OF FIGURES—Continued

FIGURE 5.16. Same as Figure 5.14 but demonstrating the behavior of descendants with a grandchild annotation. Note that the grandchild is required by its direct parent, which in turn is required by the grandparent. This result is the same even if Task 5 was short enough to end before Task 2, since its last-starting parent is Task 4.	169
FIGURE 5.17. For a set of n annotations, it is possible for each annotation to be concurrent to all other annotations.	175
FIGURE 5.18. Commonality in different Petri nets enables users to compare those points to answer fundamental research questions.	182
FIGURE 5.19. This screenshot of the Petri net visualization demonstrates the force directed layout for a single Petri net with a mouseover tooltip. . .	184
FIGURE 5.20. This screenshot of the Petri net visualization demonstrates the force directed layout for a single Petri net after editing the layout by dragging nodes.	184
FIGURE 5.21. This screenshot of the Petri net visualization demonstrates the force directed layout for two Petri nets visualized at the same time with manual node position refinement.	185
FIGURE 5.22. One possible iteration of an annotation hierarchy.	188
FIGURE 5.23. An equivalent annotation hierarchy to 5.22 with a slightly simplified representation.	188
FIGURE 5.24. An equivalent annotation hierarchy to 5.22 and 5.23, demonstrating how the same activity can be represented in many ways.	189
FIGURE 5.25. Removing “Hybrid Analysis” can simplify annotation hierarchies, as can integrating tools into other annotations rather than their own. . .	189
FIGURE 6.1. A Petri net generated from a successful <i>RevEngE CTF</i> deobfuscation challenge submission and a snippet of that submissions’s annotations.	193
FIGURE 6.2. The screenshot video view of the top participant using Binary Ninja to look through Challenge 1’s control flow graph.	195
FIGURE 6.3. The resulting Petri net from the first session of the top participant working on Challenge 1.	196
FIGURE 6.4. A timeline of the annotations provided by two analysts (left and right of the timeline) of 4 sessions from the <i>Grand Re.</i> Annotations for events that happened at approximately the same time are shown as happening simultaneously.	201

LIST OF FIGURES—*Continued*

FIGURE 7.1. The process monitor now uses a sparse approach that only records processes as they are born, die, or have significant updates. This set intersection calculation completes in $O(n)$; the intersection calculation operates on the set of current datapoints and the set of datapoints recorded in the prior process monitoring sample. All datapoints in the blue and red categories are stored, while the purple category datapoints are stored if significantly different (in CPU and/or memory use) than the previously stored datapoint from the same process.	204
FIGURE 7.2. CPU usage over time, since startup of the endpoint monitor. Environments are listed on the bottom legend; the two numbers are the interval (in seconds) to sample screenshots and process data, respectively.	206
FIGURE 7.3. Memory usage over time, since startup of the endpoint monitor. Memory is measured on the operating system as a percentage of system memory.	208
FIGURE 7.4. Percent CPU usage of individual threads when operating in the testing VM setup. Different intervals (color coded by the circles' stroke) demonstrate the performance impact of different settings.	210
FIGURE 7.5. Percent CPU usage of individual threads when operating in the testing Kali RPI4 setup.	211
FIGURE 7.6. RPI data and VM data for the main data collection threads (Screenshot generation, process monitoring, local data writing, data export, and IO reading).	212
FIGURE 7.7. The minor fault count for the virtual machine setup.	214
FIGURE 7.8. Major faults in the virtual machine setup.	215
FIGURE 7.9. Minor fault for the RPI setup.	216
FIGURE 7.10. Major faults for the RPI setup.	217
FIGURE 7.11. Amount of execution time taken by different endpoint monitor components under different environments and settings, alongside average CPU use. Except for the screenshot frame rate, these are all directly measured within the endpoint monitor. Screenshot recording and window detection proved to be too quick to measure without more precise (and computationally expensive) timers, as they measure only a single data point per metric. However, the average screenshot time here is calculated based on entire sessions data rather than internal metrics.	219
FIGURE 7.12. A zoomed in view of Figure 7.11 focusing on the screenshot frames per second metric.	220

LIST OF FIGURES—*Continued*

FIGURE 7.13. The average total time required to write each data category to disk per writeout. Note that averages are calculated only when entries to write exist; in cases like Process data larger intervals could lead to writeout periods where no data exists to write, leading to averages skewed to larger but less frequent writes.	222
FIGURE 7.14. The average time taken to write a period’s worth of data to local disk, demonstrating the superior persistence performance of the VM and interesting scaling characteristics.	223
FIGURE E.1. The administrative tables for Catalyst. The ‘TokenRequest’ table contains requested tokens for events that do not automatically approve new tokens. ‘EventPassword’ contains authentication information for non-administrators to view and annotate the subjects’ data. The ‘UserList’ contains users’ authorized tokens, which serve as authentication to download an installer. Each installer download inserts a new ‘UploadToken’ and passes it to the installer, which authenticates data uploads from the Endpoint Monitor. Diagram produced with MySql Workbench. [204]	250
FIGURE E.2. Subject data tables key from the ‘User’ table. As Catalyst streams data from subjects, the different data types map to the tables here. Data exports generally operate on these tables, though some data statistics are cached as shown in E.3. These Endpoint Monitor operates largely on identical tables provisioned to a MySql instance installed on subjects’ devices. Diagram produced with MySql Workbench. [204] . . .	251
FIGURE E.3. Additional tables provide functionality primarily for the visualization. ‘VisualizationFilters’ stores filter sets so that they might be quickly applied; ‘BoundsHistory’ and ‘ActiveHistory’ cache statistics that take too long to calculate otherwise; ‘EventPassword’ contains authentication data for tagging. Diagram produced with MySql Workbench. [204]	252
FIGURE J.1. The non-obfuscated program generated by the script in Appendix I and solved during the RevEngE CTF study. The code generating this program is about 20 KB in size.	261
FIGURE J.2. The obfuscated version of the program generated by the script in Appendix I and solved during the RevEngE CTF study. The code generating this program is about 461 KB in size.	262

LIST OF FIGURES—*Continued*

LIST OF TABLES

TABLE 1.1. A list of developments in chronological order.	32
TABLE 1.2. A list relevant software repositories containing Catalyst and Re- vEngE code.	32
TABLE 2.1. Types of code obfuscations [155].	39
TABLE 3.1. Catalyst Endpoint Monitor minimum requirements.	88
TABLE 3.2. Catalyst Server minimum requirements.	88
TABLE 4.1. Instruction counts and comparison for the deobfuscation chal- lenge broken into two tables. Instruction categories are shown by color (Arithmetic is blue, Control Flow is orange, Memory is green, and Struc- ture is yellow) and differences between the submission and original are shown as an absolute instruction count (Diff) and as a percentage of the original (Percent). Green or yellow cell coloring in these rows indicates an improvement over the original (with fewer instructions executed) and red indicates a performance decrease.	129
TABLE 4.2. <i>RevEngE CTF</i> events, durations, and prizes.	134
TABLE 4.3. The statistics regarding user participation for each event: The count of users who participated, the count of the times they downloaded the data collection software, the total number of tokens issued for the event, and the total and active time recorded, in minutes. The number of times downloaded indicates the number of devices subjects attempted to install the data collection on; the number of tokens indicates how many subjects signed up to participate in each event. Active time is defined as the sum of 5-minute intervals where there exists at least 1 user input from the keyboard or mouse.	135
TABLE 4.4. The tools used for and sizes of The <i>Grand Re</i> 's challenges. . . .	140
TABLE 4.5. The number of data points for each data category for each event, and the size on disk for the tables holding those data categories. The total size for the database is about 25 GiB.	142
TABLE 4.6. In order to attract qualified subjects for the three events in the study, each featured prize money. The amount of money and rules to win changed for each round as well; the initial RevEngE challenge featured per-problem-solved prizes while the other two offered prize money for winning the competition. In all three events, a lesser amount of prize money was offered for partially complete attempts as well.	143

LIST OF TABLES—*Continued*

TABLE 4.7. Subjects used a variety of environments and solved both x86 and ARM challenges. Ubuntu, however, was the most common operating system among subjects.	143
TABLE 4.8. Subjects solved 3 RevEngE challenges and 4 Grand Re challenges; the challenge details are listed above.	145
TABLE 4.9. Study event details and differences are listed here. Events are ordered by their start date, increasing from left to right; Catalyst maturity also, then, increases left to right. Note that "attempt" as used here refers to smaller amounts of compensation for unsuccessful submissions with valid traces showing a good faith solution attempt.	146
TABLE 7.1. Summary performance statistics running the endpoint monitor, for Kali RPI and Ubuntu VM. To ensure a uniform workload for the different tests we watched a roughly 5 minute YouTube video.	224
TABLE D.1. A summary view of subjects' activity from the Grand Re Challenge. Time is measured in minutes, while other columns are data entry counts. Usernames are anonymized automatically in Catalyst in the public data view, available at http://revenge.cs.arizona.edu/DataCollectorServer/openDataCollection/vissplash.jsp?event=GrandReChallenge&eventPassword=anondata&eventAdmin=cgtboy1988@yahoo.com	249
TABLE F.1. <i>Tigress</i> Transforms, reproduced from [307].	253
TABLE L.1. A summary view of subjects' activity from the Ghent University Assignment. Time is measured in minutes, while other columns are data entry counts.	270

ABSTRACT

Software often contains proprietary information — algorithms, intellectual property, and encryption keys, for example — which malicious actors seek to access through reverse engineering. In order to preserve the confidentiality and integrity of these assets, programmers can apply protections to their software. Code obfuscation, in particular, aims to counter reverse engineers, making asset extraction and program tampering much more difficult. In spite of decades of research into how to best generate and analyze code obfuscation and reverse engineering methods, prior efforts to model the hardness of obfuscation schemes and efficacy of reverse engineering have failed to yield robust results. This, in turn, makes code obfuscation an unpredictable protection.

The work here furthers analysis of real-world obfuscation resilience by examining reverse engineers as they overcome obfuscation in solving synthetic challenges. The general process involves (1) generating reverse engineering challenges, (2) giving those challenges to reverse engineers to solve under remote supervision, (3) collecting fine-grained traces of the reverse engineering tasks performed and (4) analyzing the resulting traces to build higher level models of reverse engineer behavior. The success of this process hinges on the validity of the challenges, the ability to attract reverse engineer subjects, the robustness of the system in gathering and analyzing generated data, and the algorithms to infer high-level attack operations from low-level trace data.

Concretely, this dissertation documents the development, deployment, refinement, and ultimately the results of using the Catalyst Data Collection System (Catalyst) to collect trace data from reverse engineers in capture-the-flag competitions,

in particular the Grand Reverse Engineering Challenge (GrandRe). Specifically, it presents (1) a methodology and system to generate basic models of human behavior remotely and asynchronously with no supervision, (2) the application of this methodology and system to reverse engineering obfuscated code, and (3) the results of that application. Alongside this, I release the reverse engineering data sets and Catalyst software for further research.

Chapter 1

INTRODUCTION

In Man at the End (MATE) attacks a malicious actor possesses a piece of software and seeks to extract an asset from that software. Typical assets include intellectual property, proprietary algorithms, encryption keys and certifications, and authentication information residing within the target program, all of which the software producer seeks to keep confidential and intact. The malicious actor uses reverse engineering techniques to obtain the asset they desire from the target program.

Software vendors try to prevent MATE attacks from succeeding by protecting their code. Code protections vary according to the exact environment they run in, but a common technique uses code transforms to obfuscate the program. In this case, a developer obfuscates program P into a functionally equivalent (but differently implemented) program P' with a set of transforms. This attempts to make the code more difficult to manually comprehend or trace or automatically analyze or manipulate, and hence makes it more difficult for malicious actors to reverse engineer the program and recover the asset.

Software developers rely on obfuscations to keep data confidential. The strength of the obfuscation scheme, then, determines assets' protection level, similar in function (though much different in form) to how the choice of cryptographic scheme determines how well a piece if data is protected against cryptanalysis. Unlike these other schemes, however, software protection researchers have yet to agree on a universal scheme for how to evaluate the protection strength offered by code obfuscation. Cryptographic schemes tend to have a rather constrained problem space, relying on

computationally difficult problems to accomplish clear-cut goals — specifically, to resist unauthorized ciphertext decryption. Metrics, then, involve determining the complexity of the computation required to do this unauthorized decryption.¹

Unfortunately, so far, no one has been able to provide clear-cut metrics for how the choice of obfuscating transformations will affect human reverse engineers' ability to break the protection. Reverse engineering relies on human intuition and know-how, using tactics, algorithms, and tools such as dynamic analysis, disassemblers, debuggers, and the construction of control flow graphs to aid the individual. As a result, evaluating the strength of code obfuscation in the face of reverse engineering requires in-depth analysis of this difficult-to-quantify human element.

Previous efforts to examine obfuscation difficulty center on normative arguments regarding program complexity and typical algorithms reverse engineers use, program metric based arguments intended to relate to reverse engineering practices, and limited human studies. Previous published approaches, however, fall short in providing an understanding of how well obfuscations hold up to unconstrained² reverse engineering attacks. Normative evaluation constrains the problem to specific and often impractical definitions which do not necessarily match reverse engineering goals and practices; software complexity metrics have not been empirically linked to actual reverse engineering difficulty; available human studies have been constrained in choice of challenge problems and diversity and number of human subjects.

The work presented in this thesis aims to build on prior human studies by introducing methods and tools to more precisely analyze reverse engineers as they

¹Side channel attacks on cryptographic implementations, however, more closely parallels code obfuscation and reverse engineering due to its reliance on human intuition in finding ways around the hard problems rather than solving them outright.

²Prior work demonstrates the efficacy and subsequent hardness of specific attacks against specific protections, but in practice attackers are not constrained to these specific methods.

counteract specific software protections.

Ultimately, my goal is to study expert reverse engineers solving well-designed synthetic reverse engineering problems (which aim to mirror real-world applications) with data collection tools that help inform the hardness of practical code protections and efficacy of different reverse engineering practices. In this endeavour, I was able to gather a significant amount of data from reverse engineers solving synthetic challenges; analysis of this data proves difficult but initial visualization enables dataset annotation, which has enabled initial attempts to build models from the data. This limited analysis provides some insight into reverse engineering practices and future work building upon these efforts, particularly in the annotation stage, can potentially produce more robust results.

1.1 Experimental Methodology

Human-based reverse engineering studies hold potential to gain insights into the strength of obfuscating transformations and to reveal which methods counter them most effectively, as these studies capture the human element on which reverse engineering relies. However, prior studies produced data limited by geographic isolation, subject expertise, granularity of the collected human-device interaction (HDI) data, and complexity of challenge problems. Where other studies qualitatively studied subjects as they engaged in reverse engineering in smaller-scale, localized experiments within specific groups — such as students in a classroom setting — I seek to source detailed, quantitative data from diverse groups of subjects as they reverse engineer asynchronously around the globe with the hope that this will introduce larger scale, additional subject method stratification, and finer detail in results.

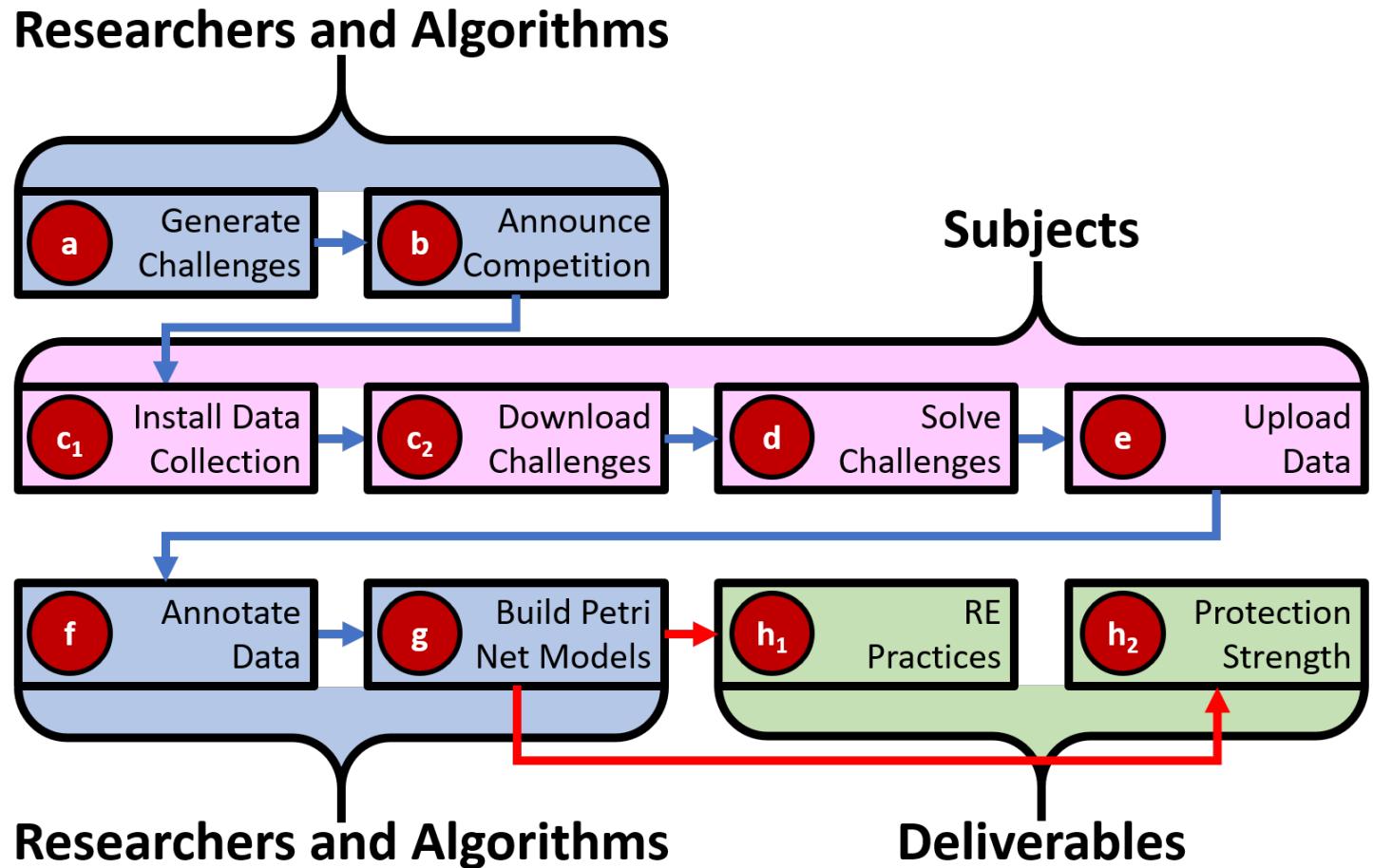


Figure 1.1: The big picture: Researchers generate challenges which experts solve while generating traces for analysis.

In order to do this, I built a system to collect HDI data from subjects as they solve challenges consisting of reverse engineering tasks on obfuscated binaries. I further built visualization, analysis, and modeling tools into the data collection framework to evaluate the low-level data collected from subjects. I then, along with a team of researchers from The University of Arizona, Ghent University, and Irdeto, offered monetary rewards for solving these challenges while using the data collection software.

The overall methodology's steps (illustrated in Figure 1.1) are:

- (a) **Reproduce** real world reverse engineering problems by *generating representative challenges*.
- (b) **Recruit** *subjects* to participate in challenge solving.
- (c–e) **Record** the subjects as they solve challenges using the *endpoint monitor* and *back end*.
- (f) **Review** the collected data by annotating the dataset within the *visualization*.
- (g) **Resolve** the annotated data into reverse engineering models with the *Petri net generator*.
- (h_{1–2}) **Relate** these models to code protection strength through *data-driven metrics*.

1.2 Research Questions

This work addresses 5 research questions associated with understanding how reverse engineering techniques work — the tools and techniques typically used, the require-

ments and efficacy of those strategies, and how well they counter code obfuscation. These questions are listed below and referenced throughout the thesis as RQ1-RQ5.

RQ1: Is it possible to build a system to remotely and unobtrusively collect high fidelity HDI data from experts in their own heterogeneous computing environments as they reverse engineer software?

Collecting trace data from reverse engineers as they solve problems enables quantitative, fine-grained analysis of their methods. This is not possible to do remotely and asynchronously without appropriate data collection software. This software must collect the data with sufficient fidelity that it will allow further research questions to be answered.

Software collecting data from subjects firstly must function in the subjects' computing environments allowing them to use the tools and methods with which they are familiar. The subjects here are experts in software reverse engineering; these experts may use a variety of platforms, from cloud services to Raspberry PIs. Generally, these experts use major architectures such as x86 and ARM and major operating systems including Linux variants, Windows, and MacOS; the data collection software must support these environments and, ideally, be easily portable to other environments as the need arises.³

In addition to functioning correctly and collecting HDI data, the software must be performant in several areas to meet this goal. On the endpoint data collector, the software must use few enough resources and perform fast enough without causing system lag that reverse engineers find it acceptable to use, given rewards offered for doing so. On the server side, the software must be efficient enough to ingest

³In one early study attempt, a well known reverse engineer — Rolf Rolles — replied to the announcement that he would only work in a Windows environment, which our system did not support at the time.

data from multiple (on the order of 100 challenge participants) subjects running the endpoint software concurrently and export that data to analysts (on the order of 10 concurrent users) viewing the data concurrently, considering the server hardware constraints.

The visualization and analysis component must be available for researchers to easily access in their own uncontrolled computing environments. Further, it must perform well enough to maintain usability as data scales — meaning researchers can use the visualization to analyze data without any problems as more data is collected from subjects.

The server must also be secure and anonymous in order to satisfy institutional review board (IRB) human subject research requirements and generally encourage computer security experts to use the data collection software. In anticipation of reverse engineers' unwillingness to divulge their identities, data exported by the server must be anonymized such that it does not deter participation. Additionally, the endpoint monitor must not expose sensitive data (like users entering passwords) to the world at large.

I built a system that remotely collects HDI data and deployed it during several reverse engineering events. These resulted in over 130 hours of active trace data. A follow up event done by Ghent University collected an additional 69 hours of active trace using students. Performance analysis and feedback from participants demonstrates that the data collection software runs unobtrusively.

RQ2: What type of incentives are needed to recruit a significant number of expert reverse engineer to participate in these studies?

Reverse engineers may choose not to participate in monitored studies in order to protect their proprietary methods or because there exists high, well paying de-

mand for their expertise already. Rewards, ideally, overcome these barriers. Reward efficiency is defined as the amount and quality of data produced for the amount spent; the rewards offered succeed if they — along with sufficient promotion efforts and strategies — generate interesting results while remaining within the project’s budget.

The incentives offered to reverse engineers generated a significant amount of participation — 123 event sign ups with 44 users generating trace using the data collection software.

RQ3: Will generated synthetic reverse engineering challenges capture real-world reverse engineering tasks?

Real-world reverse engineering tasks often target sophisticated, large-scale software with complicated attack goals. In a capture the flag (CTF)-like setting we must use smaller-scale challenges, and this runs the risk of generating data not representative of real-world reverse engineering. Moreover, measuring how well the challenges correspond to real-life situations presents a problem in itself, given the variety and proprietary nature of software protection and reverse engineering. Nonetheless, the work presented here proposes several types of reverse engineering challenges with code protections based on researchers’ conception of typical real-world situations and methods, and some of our challenges are produced by tools from the leading company in this space, Irdeto.

The reverse engineering events used challenges generated by industry professionals and researchers, using tools employed in the field, though further analysis could further compare the synthetic challenges to real world experience.

RQ4: Can the resulting datasets be lifted to human-understandable models of subjects' behavior?

The HDI data collected from reverse engineers is low-level: It includes input data such as keyboard and mouse clicks, as well as window information and screenshots. At this level, the data does not provide the information we seek regarding reverse engineering practices and code obfuscation difficulty. Rather, pre-processing of this data, lifting it to a higher semantic representation, can facilitate further analysis. The research here presents one type of analysis and model-building that attempts to accomplish these goals; we are releasing our raw data to enable other researchers to develop further analyses and modeling techniques.

Initial attempts to annotate the data and reviews thereof show similar understanding of the underlying data between analysts, but annotations thus far have been slow to generate and inconsistent in form. Further work in annotation (and coding practices) is necessary.

RQ5: Do these analyses and models provide valuable insight into reverse engineering practices and code obfuscation efficacy?

Ultimately, this research aims to provide an empirical basis with which code obfuscation strength can be measured, as well as an understanding of the reverse engineering methods used to defeat different types of obfuscation. Metrics regarding those practices include how long a reverse engineer takes to solve reverse engineering tasks, the methods and tools used, and the computing environments and resources required; discerning these metrics from the collected data and classifying it according to obfuscation type defeated provides the necessary empirical basis to answer this research question.

The limited-scale, initial models generated from annotations (and the underlying data itself, when visualized) show reverse engineers using known methods to solve challenges. The data also provides metrics (such as keystrokes or time taken) which demonstrate the time and effort required for these strategies. However, more work on RQ4 is necessary to generate more annotations for analysis and, importantly, comparison between multiple subjects.

1.3 Current Results

To gather, curate, and analyze HDI data from subjects, I built the Catalyst Data Collection Engine (CDCE or Catalyst). This system runs unobtrusively even on average performance commodity hardware. Specifically, it **records** data from subjects using the *endpoint monitor* and the *back end server*, **reviews** the collected data by annotating it in the *visualization*, and **resolves** the data by applying a *Petri net generation* algorithm to annotations.

Using CDCE, a international research team and I hosted reverse engineering challenges. The initial research team included myself and Christian Collberg from the University of Arizona for the RevEngE and RevEngE CTF challenge sets; these sets featured challenges generated with the *Tigress* obfuscator. The team then expanded to include collaborators from Ghent University and Irdeto, a private company, for the last challenge set — the Grand Reverse Engineering Challenge. This challenge set included not only *Tigress*-generated problems, but also challenges from Ghent University obfuscated with ASPIRE and Irdeto challenges protected with their proprietary obfuscation. A summary of these events, in chronological order, may be found in Table 1.1; a summary of associated code repositories may be found in Table 1.2.

Year	Item	Abbreviation	Description
2016	Catalyst Data Collection Engine	CDCE or Catalyst	Built
2016	Reverse Engineering Engine teaching tool	RevEngE Teaching Tool	Built, deployed, integrated CDCE
2017	Reverse Engineering Engine Competition	RevEngE	Expanded RevEngE Teaching Tool
2020	Reverse Engineering Engine Capture the Flag	RevEngE CTF	Adapted RevEngE
2021	Reverse Engineering Engine Study	RevEngE Study	Follow up with RevEngE CTF subjects
2021	Grand Reverse Engineering Challenge	Grand Re	Built with team and CDCE

Table 1.1: A list of developments in chronological order.

Repository Name	Description	Link
CybercraftDataCollectionConnector	The endpoint monitor component for Catalyst.	https://github.com/taylor239/CybercraftDataCollectionConnector
UserMonitorServer	The Catalyst web server, including visualizations.	https://github.com/taylor239/UserMonitorServer
RevEngECompetition	The Reverse Engineering Engine competition webapp.	https://github.com/taylor239/RevEngECompetition
RevEngE	The educational version of RevEngE.	https://github.com/taylor239/RevEngE
catalystSnap	A snapshot of Catalyst, RevEngE, and related documents.	https://github.com/taylor239/catalystSnap
Data Archive	The data collected from RevEngE and The Grand Re.	https://doi.org/10.25422/azu.data.19858645

Table 1.2: A list relevant software repositories containing Catalyst and RevEngE code.

These challenges all employed a variety of code obfuscation types and transforms in order to determine how easily each might be overcome. By incentivizing participation with over \$10,000 in monetary rewards and announcing challenges on social media and email lists, we successfully saw over 100 participants sign up, of which over 40 were successful in running the data collection software.

The endpoint collection performed adequately enough that these participants logged over 130 sessions with over 130 hours of activity — a total of over 25 GiB in the back end’s storage. Subjects ran the data collection primarily on x86_64 devices with an Ubuntu operating system, but some subjects also ran the software on x86_64 on Windows or Kali Linux and arm_64 with Raspbian; Ubuntu on x86_64 provided the most stable platform for participants; in these environments, the endpoint monitors consistently consumed about 100% of a single CPU core and 15% of system memory. The back end received, stored, and served this data to the visualization without issue. This informs RQ1.

Ultimately, 3 participants solved a total of 10 discrete challenges. The subjects who succeeded in solving challenges recorded about 65 hours of activity. The winning subjects received a total of \$8,500 in prizes. These results inform RQ2 since the prizes resulted in a significant amount of participation from professionals who could solve challenges and To a lesser extent, this also informs RQ3 since the participants had enough real-world reverse engineering experience to solve these challenges using known tools and methods employed in the field.

After the challenges finished, the research team used the visualization tool to annotate the dataset, attempting to generate Petri nets with the resulting annotations. While this analysis work is ongoing, initial results demonstrate that researchers can discern the methods and tools a subject uses to reverse engineer and translate this into annotations. With these annotations, the algorithm presented in this disser-

tation automatically generates Petri nets, demonstrating one (of potentially many) answer(s) to RQ4. Because these Petri nets map to underlying data, they include the metrics presented in RQ5 and provide a basis to evaluate obfuscation difficulty against the attacks subjects used to defeat them.

So far in this analysis, the research team has observed subjects using dynamic analysis techniques with debuggers and (statically generated) control flow graphs to solve challenges. The subjects used Binary Ninja, Ghidra, and Ida tool kits to do this, as well as other tools such as GDB, Triton, Visual Studio Code, and Python. The winners all used x86_64 environments running Ubuntu or Kali Linux; for challenges which were available only on ARM/Linux, they opted to use emulators (such as QEMU) rather than directly operating in an ARM environment. The winning participants also reversed virtualization obfuscations in particular by writing interpreter scripts to translate the virtualized functions into simpler code.

One winner blogged about the experience, giving further details on how they approached solving the challenges:

<https://binary.ninja/2021/09/02/winning-the-grand-re-challenge.html>

This research also raises questions about the limits of current reverse engineering techniques as well as the quantity, quality, and value of reverse engineers: Of the 50 RevEngE and 10 Grand Re Challenge problems generated, a total of 3 participants submitted only 10 individual solutions — solving a total of only 7 discrete challenges. It is not clear why, in a reasonable size field like reverse engineering, more participants did not solve challenges. Potential reasons for this range from data collection software difficulties to privacy concerns to proprietary method protection, simply lacking the ability to solve challenges, or not having enough of a prize to be worthwhile. The reasons could also be something else entirely. Exit surveys did not generate responses, so these questions may need to be examined in future work.

1.4 Dissertation Overview

This dissertation first reviews related work in Chapter 2, then answers RQ1 by detailing the design and implementation of Catalyst in Chapter 3. Chapter 4 catalogs reverse engineering studies (with Catalyst deployments) that the research teams and I hosted, providing insight into RQ2 and RQ3. Chapter 5 describes the Catalyst data analysis and visualization components, and Chapter 6 demonstrates those tools' application to the data collected from the reverse engineering studies; these chapters inform RQ4 and RQ5, respectively. Chapter 8 recommends future work that can enhance the answers this dissertation offers for all of the research questions, and Chapter 9 summarizes the work and its subsequent conclusions with regards to each research question.

Chapter 2

BACKGROUND AND RELATED WORK

Man-at-the-end attacks occur when an adversary gains control over a piece of software and attempts to extract an asset from it [7]. That is, the attacker attempts to violate the confidentiality and/or integrity of assets in software, such as copyrighted materials, embedded keys and records, and algorithms. A well-known example is software “cracking” — avoiding having to pay licensing fees by hacking the application to obtain a license key [95]. Attackers use a variety of *reverse engineering* techniques to extract assets from programs. To counter this, vendors apply *software protection* techniques to their programs.

In this dissertation, I monitor reverse engineers as they solve synthetic problems in order to model their expertise. The research presented here builds on previous work in six areas:

1. Software protection against man at the end attacks (discussed in Section 2.1) which this dissertation attempts to analyze.
2. Tools to protect software against MATE attacks (Section 2.1) which implement the methods described above and are useful in analysis efforts.
3. Reverse engineering attacks to compromise protected software (Section 2.3) which counter software protections; analyzing these attacks in turn analyzes code protections.
4. Ways to evaluate software protection techniques in light of available reverse engineering attacks (Section 2.4) which the analysis efforts detailed in this

dissertation expand upon.

5. Ways to model attacks and defenses (Section 2.5) as potential tools in analyzing reverse engineering attacks and their relationships to software protection techniques.
6. Ways to build models of of human behavior through observations (Section 2.6) which can potentially inform attack and defense analysis by studying software reverse engineers.

2.1 Software Protection Techniques

Much has been written about software protection and reverse engineering. For example, Ahmadvand et al. [5] provide a thorough taxonomy of well known code obfuscation and reverse engineering techniques. That taxonomy describes different types of software protection techniques and reverse engineering methods to counter them, as well as current techniques used in evaluating the strength of different protection schemes, including various types of code obfuscation. The *Malware Reverse Engineering Handbook* [25] catalogs many of these methods and lists typical tool implementations used within the context of malware analysis. In this and the next section I will explore these topics further.

In order to protect against MATE attacks, a software vendor/author might apply software protection techniques to prevent or delay an attacker from violating the confidentiality and integrity of assets in their program. Several types of software protection exist, ranging from platform-independent, source-level, code obfuscation techniques to platform-specific, binary-level tamperproofing techniques. Often, software vendors use a combination of protection techniques to leverage each one's unique strengths.

This dissertation attempts to examine and quantify the strength of these protections, both alone and in combinations. This analysis, in turn, could aid software vendors in selecting the most effective protections and provide an estimate of protected assets’ lifespan before reverse engineers break those protections. While many protection techniques exist, examining as many as possible provides the greatest amount of insight. Some of the more common code protection schemes are thus discussed next; the reverse engineering events ultimately included challenges with a subset of these techniques.

2.1.1 Code Obfuscation

Code obfuscation transforms an original program P into a functionally equivalent but differently formed program P' where P' is, ideally, more difficult for humans (or humans assisted by machines) to extract asset(s) from. Essentially, obfuscation utilizes *security through obscurity*, which is normally a shunned design choice for “truly secure” systems [149]. However, when an adversary has complete control over a piece of software in their own, foreign, unknown environment, obscurity tends to be the only option available¹ — and it is unclear whether that protection is actually weak in the context of code obfuscation (see Section 2.4) despite theoretical arguments to that effect. Collberg et al types [96] formalized code obfuscation, provided criteria for evaluating protection levels, and enumerated several transform.

Code obfuscation may be done on many different types of code in different stages of a build pipeline — from high level Javascript interpreted source code [182] to intermediate representations like Java bytecode [190] and LLVM IR code [83] and low level machine code [123]. Obfuscation also uses a variety of code transform

¹Other techniques tend to require architecture changes introducing components such as custom hardware or server side execution away from the end user [120].

types, which generally focus on either control flow [156] or data structures [348]. A third category, “Layout” transforms employ more lightweight changes, such as removing variable names and randomizing addresses without changing control flow. Hosseinzadeh et al. [155] provide a survey of common transform types and techniques by evaluating over 300 research papers as of 2018. These are shown in Table 2.1.

Table 2.1: Types of code obfuscations [155].

Control Flow	Data	Layout
<ul style="list-style-type: none"> • Opaque Predicates • Bogus Insertion • Polymorphism • Branching Functions • Inlining Methods • Self-Modification • Instruction Transformation • Reordering Blocks • Jump Table Spoofing • Code Transformation • Loop Transformation 	<ul style="list-style-type: none"> • Class Transformation • Array Transformation • Variable Transformation • Code Substitution • Encryption 	<ul style="list-style-type: none"> • Remove Debugging Information • Identifier Renaming

2.1.2 Code Diversification

Code diversification attempts to create variability in distributed copies of software [201]. Diversification operates similarly to code obfuscation — it transforms the source program P into a set of (rather than singular, as in obfuscation) functionally equivalent copies P'_1 , P'_2 ... P'_x prior to distributing these copies. This introduces difficulty in creating a uniform exploit to attack a program, as each actual program instance has differing internal structure [100]. Diversification methods can also be used to watermark code [365]. While code obfuscation methods like virtualization can be used to introduce diversity into a program [16], other more specialized methods like internal interface diversification can introduce protective diversity only in specific areas to protect against malware [266].

2.1.3 Tamper Proofing

Tamper proofing seeks to prevent attackers from changing a program's behavior, which would allow them to execute the program in an unintended way. Checksumming algorithms tamperproof programs by computing one or more hashes on itself to detect any changes [237]. This method relies only on commodity hardware, with no external resources, unlike methods relying on specialized hardware such as tamper-proof hardware tokens [138] or trusted computing bases² [223, 356]; network-based systems also promise a level of tamperproofing, but rely on active connections between endpoints or on established trusted computing bases — as is the case with *remote attestation* [228, 140, 324]. While checksum systems tend to be the most versatile tamperproofing, they themselves can be defeated with reverse engineering and tampering; using these systems in conjunction with obfuscation strengthens them,

²Trusted computing bases provide cryptographic modules which can be used as roots of trust (ROT) which can ensure the integrity of software execution [119].

but only to the extent that the obfuscation’s strength provides [261, 129].

2.1.4 Whiteboxing

Whitebox security/cryptography is a software protection paradigm in which a program embeds an encryption key that must be protected from users [48, 55]. The key is subsequently used to encrypt and decrypt sensitive assets — sometimes even the program itself through encryption-based obfuscating transforms. Software protections aim to preserve the confidentiality of this key, often doing so with obfuscation techniques [343, 340]. Thus, a decryption function with a visible key $D(key, data)$ instead becomes $D(data)$ with an opaque *key*; the aim then becomes to protect the *key* asset in the program which is used internally in the decryption function.

Unlike other scenarios, whitebox security does not necessarily require perfect confidentiality; in digital rights management cases, for instance, tracability³ techniques such as watermarking⁴ let rights enforcers to track down the man at the end who broke the whitebox [109].

2.2 Code Protection Tools

During the course of the research presented here, several rounds of studies employed binary reverse engineering challenges protected with these methods, and in particular obfuscated with a subset of the transforms listed in Section 2.1.1. The challenges utilized existing tools including the Tigress Obfuscator; the ASPIRE Framework and its included Diablo binary rewriter; and proprietary tools from Irdeto, a private company. These tools have all been used in commercial and research applications,

³Tracability in this context refers being able to identify the particular program copy wherefrom leaked data originates.

⁴Watermarking involves adding identifiable data to a unique piece of intellectual property [97].

which helps ensure that the protections they provide represent those reverse engineers find in the field.

2.2.1 Tigress

Tigress is a C language, source to source code obfuscator written by Christian Collberg at the University of Arizona [94]. It includes a program generation function which generates a simple hash function type of program with the option to encode simple assets therein [309]. Though free to use for research purposes, its author does not release its source. Researchers have used Tigress to generate test cases for obfuscation and reverse engineering research [316, 151, 26, 171, 302, 186, 315]. Early rounds of the research studies presented here exclusively used Tigress as the program generation and obfuscation components in generating challenges [309, 309, 307]. Tigress supports several obfuscating transform types, including those in Appendix F. Figures 2.1, 2.2, and 2.3 show a simple program, a set of transforms in a Tigress script, and an obfuscated version of the simple program, illustrating how Tigress works. Figures 2.4 and 2.5 (both reproduced from prior work [307]) demonstrate a virtualization transform on a simple program with the Tigress obfuscator.

```

1 #include <time.h>
2
3 void init () {}
4
5 int main() {
6     int x;
7     x++;
8 }
```

Figure 2.1: A simple C program.

```
tigress --Seed=42 in.c --out=out.c \
--Transform=EncodeArithmetic --Functions=main \
--Transform=Virtualize --Functions=main --VirtualizeDispatch=direct \
--VirtualizeSuperOpsRatio=2.0 --VirtualizeMaxMergeLength=5 \
--Transform=SelfModify --Functions=main --SelfModifyFraction=1 ...
```

The diagram illustrates the use of the tigress command with several arguments. Five orange callouts point to specific parts of the command line:

- 1st Transform**: Points to the first argument `--Transform=EncodeArithmetic`.
- 2nd Transform**: Points to the second argument `--Transform=Virtualize`.
- 3rd Transform**: Points to the third argument `--Transform=SelfModify`.
- Target function**: Points to the argument `--Functions=main`.
- Transform variant**: Points to the argument `--VirtualizeDispatch=direct`.

Figure 2.2: An example set of tigress arguments which obfuscate a given file with encoded arithmetic, virtualization, and self-modifying code.

```

1 union node { int _int; };
2 int main( ) {
3     char vars[32]; union node stack[32];
4     union node *sp; void **pc;
5     void *bytecode[1][15] = {{
6         &&local_load_const_xor, (void *)24UL, (void *)1UL,
7         &&local_load_const_or_const, (void *)24UL, (void *)1UL, (void *)1
8             UL,
9             &&shiftl_minus_local_store_const, (void *)24UL, (void *)0UL,
10            &&local_store, (void *)28UL,
11            &&local_load, (void *)28UL,
12            &&return_int }};
13     sp=stack; pc=bytecode; goto *(pc);
14     local_store:
15         pc++;
16         *((int *)((void *)vars+*((int *)pc)))=(sp+0)>_int;
17         sp--; pc++; goto *(pc);
18     local_load_const_or_const: pc++;
19         (sp+1)>_int=*((int *)((void *)(vars+*((int *)pc))))|*((int *)(
20             pc+1));
21         (sp+2)>_int=*((int *)((pc+2))); sp+=2; pc+=3;
22         goto *(pc);
23     local_load:
24         pc++;
25         (sp+1)>_int=*((int *)((void *)(vars+*((int *)pc)))); 
26         sp++; pc++; goto *(pc);
27     return_int: pc++;
28         return ((sp+0)>_int); goto *(pc);
29     local_load_const_xor: pc++;
30         (sp+1)>_int=*((int *)((void *)(vars+*((int *)pc))))^*((int *)(
31             pc+1));
32         sp++; pc+=2; goto *(pc);
33     shiftl_minus_local_store_const: pc++;
34         (sp+-2)>_int=((sp+-1)>_int<<(sp+0)>_int)-(sp+-2)>_int;
35         *((int *)((void *)(vars+*((int *)pc))))=(sp+-2)>_int;
36         (sp+-2)>_int=*((int *)((pc+1))); sp+=-2; pc+=2;
37         goto *(pc);
38 }

```

Figure 2.3: The simple program shown in Figure 2.1 after obfuscation with the tigress script in Figure 2.2.

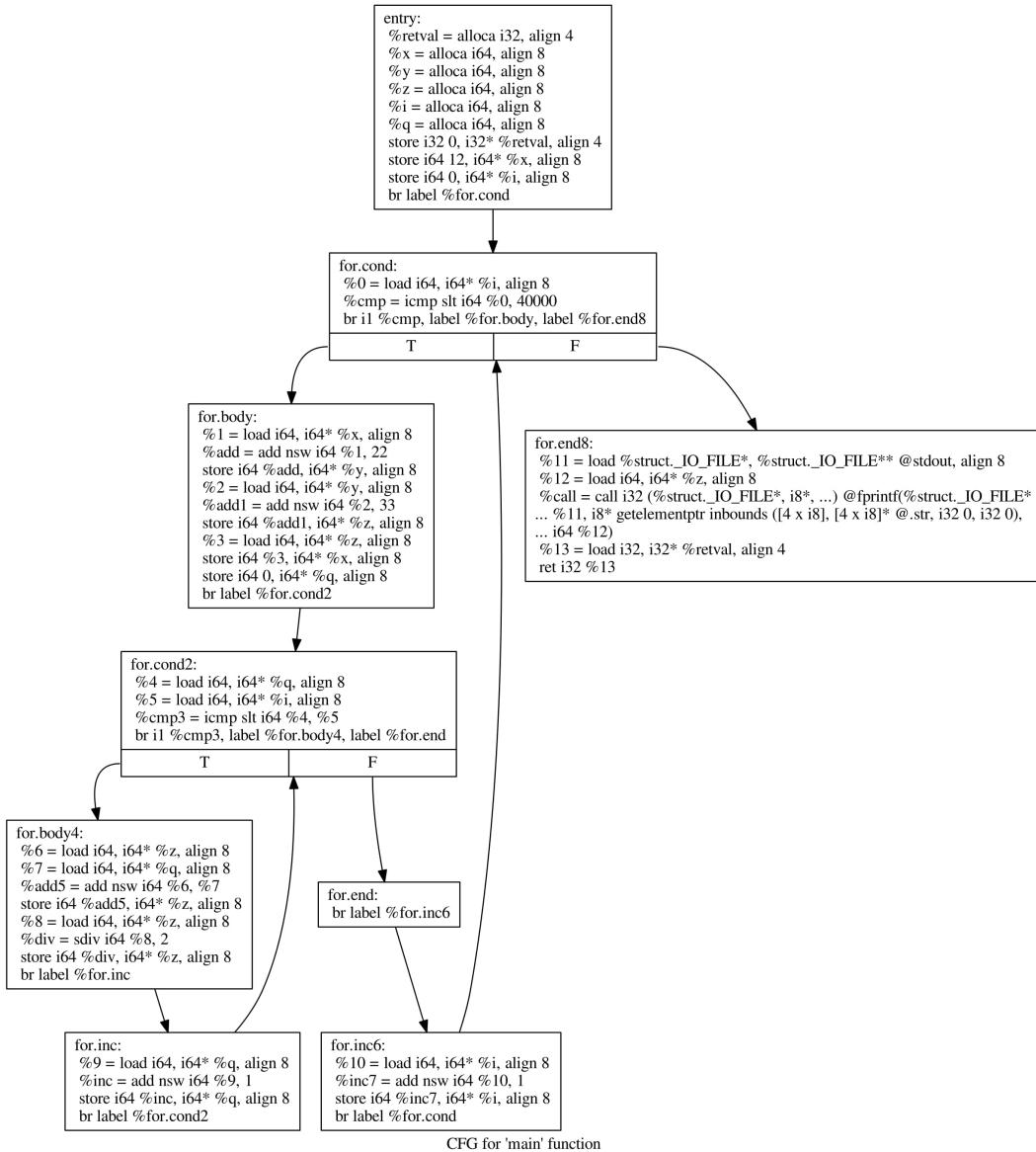


Figure 2.4: The control flow graph here derives from a simple, sample program with a single loop, a few arithmetic operations, and a print function at the end. LLVM [200] compiled this program and generated the static control flow graph.

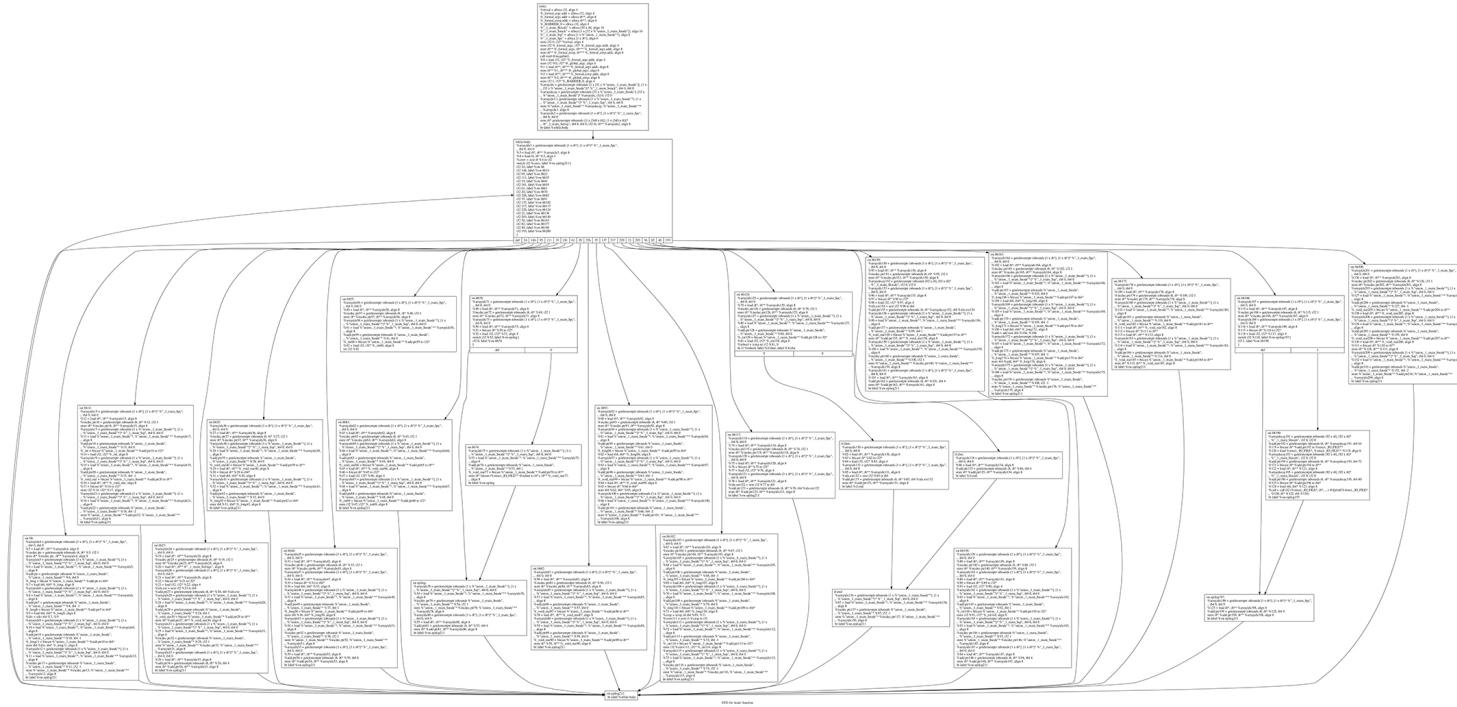


Figure 2.5: This control flow graph resulted from obfuscating the program in Figure 2.4 with a virtualization transformation. Virtualization — a type of instruction transformation — transforms blocks of code into virtual instructions; a dispatcher (the most connected block near the top in the center) delegates control to these virtual instructions. During execution, the virtual instructions executed functionally mirror the non-obfuscated execution; ideally, this virtualized control flow is more difficult for reverse engineers to analyze.

2.2.2 ASPIRE

A consortium consisting of Ghent University, Fondazione Bruno Kessler, Gemalto, Nagravision, POLITO, SFNT Europe GmbH, and The University of East London developed the ASPIRE (Advanced Software Protection: Integration, Research, and Exploitation) Framework [106, 34]. ASPIRE allows software developers to annotate programs with information which the ASPIRE compiler toolchain and decision support system then use to apply transforms during the project build. The process aims to abstract low level protection techniques from high level design details regarding where a program needs protection, as well as provide metric-based autotuning to find optimized protection schemes.

ASPIRE operates primarily on C source code. It includes a wide range of software protections that operate in source-to-source, binary-to-binary, and mixed modes; as well as components that introduce more intricate architecture such as client-server communication to move code out of local devices. The ASPIRE team amassed these protections by building an open framework and integrating tools such as the Diablo link-time rewriter [108, 320]. Notably, it focuses on five broad capabilities:

1. Data hiding with whitebox techniques and code obfuscation
2. Code hiding with obfuscation and server side execution
3. Anti-tampering with code checking and anti-debugging methods
4. Remote attestation of the anti tampering checking
5. Renewability with code diversification.

ASPIRE is free to use and has been used to generate test cases for reverse engineering and code protection research [78, 359, 324, 35]. The research here employs

ASPIRE to generate challenges for The Grand Re.

2.2.3 Irdeto

Irdeto (previously Cloakware) is a cybersecurity firm working primarily in the field of endpoint software protection [164]. They offer whitebox cryptography, anti tampering, and code obfuscation with their proprietary toolsets [217, 209]. Researchers at Irdeto pioneered whitebox security, developing one of the first attempted whitebox-safe cryptographic implementations [92]. The company recently developed heuristic and machine learning models to analyze unknown binaries to automatically determine their level of security [107]. Irdeto used some of their tools to generate challenges for The Grand Re.

2.3 Reverse Engineering Techniques

This dissertation examines code protection techniques by analyzing reverse engineers' attacks against them. Knowledge of the basic tools and methods used in reverse engineering is important in examining what reverse engineers do to solve reverse engineering challenges.

In the context of man-at-the-end attacks, reverse engineers have inconsistent goals. In some cases, they might attempt to extract some data from a program. In other cases, they may want to deobfuscate code, generating a human-understandable part or whole of a program from an obfuscated one [98]. In yet other cases, the reverse engineer may wish to generate signatures for malware detection [15]. In every case, the attacker seeks to obtain an asset from a program — where that asset could be the target of any of the listed goals [35, 104]. Reverse engineering methods have similar variability, particularly in automation level and in operating principle. Their spectrum of automation ranges from completely automated deobfuscating unpack-

ers [272] to engineer-guided control flow graph traversal [262, 169] to looking at raw code directly [99]. These methods may, however, be generally categorized as static, dynamic, or symbolic based on their operating principles.

Often, tools from all of these categories are included in larger reverse engineering frameworks, such as Binary Ninja [252, 258, 224] and Ghidra [271, 326, 224]. Sometimes, tools are even ported into these frameworks, as when Binary Ninja integrated and expanded upon GDB, LLDB, and DbgEng debuggers in order to introduce robust debugging infrastructure into its framework [198]. Additionally, some tools and methods fall outside of these categories, such as the SensorRE [147] which tracks provenance as reverse engineers do things to their programs of interest.

2.3.1 Static Analysis

Static analysis involves analyzing a program without running it [68]. Typical applications of static analysis include static disassemblers [18], static control flow graph generators [240] and associated visualizations [112], and static decompilers [84]. Automated reverse engineering tools using static analysis exist as well [36]. Static analysis suffers from several limitations: Because instructions on x86 and typical CISC architectures (though notably not on ARM and RISC architectures) are not of fixed length [47], static disassemblers have difficulty in determining where one instruction ends and another starts; neither linear sweep nor recursive traversal disassembly methods work perfectly and static-heuristic approaches are not foolproof [192, 213, 54]. Miller et al. [234] demonstrate an improved but imperfect solution to these issues by pruning instructions generated with superset disassembly. Superset disassemblers generate one instruction per byte; the resulting output is a set of all potential instructions in a program even though not each instruction produced is reachable [37]. Methods to statically generate control flow graphs suffer

from similar issues — jump instructions, in particular, can cause issues with indirect operands which cannot be easily predicted in a static context [312]. In short, where aspects of the program structure itself rely on user input, static analysis tends to fail. Most often, static analysis is employed alongside dynamic analysis, given these limits [173].

2.3.2 Dynamic Analysis

In contrast to static analysis, dynamic analysis involves running a programs to determine how it behaves [304]. While static analysis easily covers the entirety of a program, dynamic analysis only examines the program execution — which generally does not cover the entire codebase; this becomes problematic when an attacker wishes to uncover program states hidden within a program — such as executing at all if a license key is required — as typical program runs ought not reach those states [367, 102]. Typical tools include debuggers like those included in Ghidra [118], Ida [117], Ollydbg [111], radare2 [321], GDB [297], and WinDbg [1].

Dynamic binary instrumentation has emerged as an alternative to traditional debuggers, with similar function but a differing operating principle involving just in time compilation [226]. Sandboxing environments in traditional software solutions [263], virtual machines [269], and emulators like QEMU [38] all attempt to isolate dynamic analysis from corrupting devices as well as provide tooling to gather data to analyze.

Often, reverse engineers use these tools alongside statically generated disassembled code or control flow graphs; the statically generated components provide a view of the whole program, while the dynamic analysis with debuggers allows the reverse engineer to examine paths execution takes through it. Additionally, where some static tools like disassemblers and control flow graph generators fail, often a dynamic

version of the same tool can be applied for the portion of the program executed, such as disassembling the instructions executed with a dynamic disassembler [238].

More automated tools such as Cuckoo [333] run programs to collect data and analyze it. Cuckoo itself runs pieces of suspected malware in a sandbox environment, analyzes what that program does to resources such as the filesystem or network activity, and provides a report to malware analysts. Live, dynamically collected memory images and allocations are used in many ways: Choi et al. [91] demonstrate a method to analyze API calls through dynamic memory analysis while Mercier et al. demonstrate a method to analyze data structures from memory as well [233]. Binkley et al. [44] show how to remove unnecessary code (such as that introduced by obfuscation) with observational program slicing. This is done by removing a statement (or multiple statements) from a running program to determine if that statement changed execution; static program slicing methods can provide heuristics regarding which instructions to change.

2.3.3 Symbolic Execution

An enhancement to traditional dynamic analysis, symbolic execution involves using symbolic variables to trace paths through a program and examine its control flow [296]. Symbolic variables do not have a concrete value, unlike in typical dynamic analysis; the variables can be any value, constrained only by the reverse engineer. Like dynamic analysis, the program “executes” with these symbolic variables, tracing how that variable effects control flow and finding paths through the control flow graph based on branching instructions; the tool klee [62] provides an example of symbolic execution designed to find test cases to cover all (or at least most) execution paths in a program. Salwan et al. [278] used symbolic execution in order to reverse engineer programs obfuscated with Tigress virtualization, transforming the obfuscated code

into code functionally equivalent and similar in size to the unobfuscated code.

2.3.4 Concolic Execution

Concolic execution presents a strategy to limit the computational complexity of symbolic analysis. While symbolic execution typically makes all program input values symbolic, concolic execution makes some of the variables symbolic while making others concrete [352]. Where simple symbolic execution slows, such as loops that can be made infinite depending on variable values, concolic execution can find a path through by using a concrete value [45]. In cases like this, symbolic execution reaches the looping branch(es) and finds ranges that either repeat or leave loop to execute elsewhere in the program. If a symbolic variable can cause an infinite loop, the symbolic execution will continue to reach this state over and over.

Yadegari et al. [351] demonstrate an automated deobfuscation method using concolic execution combined with taint analysis. Popular manual tools employing concolic execution include angr [329], Triton [281], and BAP [58]. These concolic execution engines suffer from issues related to accuracy and scalability in an analysis using synthetic problems from Xu et al [349].

2.4 Evaluating Obfuscation Strength

This dissertation expands on prior efforts to evaluate software protection — and obfuscation in particular — techniques. Successfully doing this allows software vendors to select maximally effective techniques and estimate how well assets will last after distribution.

Code obfuscation serves as the most versatile software protection technique. However, determining the level of protection offered by any particular obfuscating transform or combination thereof is a difficult problem. Collberg et al. [96] describe the

strength of a given obfuscation set as a combination of three factors:

- Potency — how well is the code protected from human understanding?
- Resilience — how well is the code protected from automatic analysis?
- Cost — how much overhead is added by a given obfuscation?

Cost is not explicitly considered in the analysis in this dissertation⁵ and does not have bearing on the obfuscating strength of a given obfuscation — though it does effect the efficiency of it. Moreover, potency and resilience here are considered to be components of the same thing: Software reverse engineers use both manual and automated methods to compromise code, and the extent to which the obfuscation stops this is its *absolute strength*. Additionally, when used in conjunction with tamperproofing, the *attack resistance* must also be considered as a product of both the obfuscation’s absolute strength and the inherent resistance of the tamperproofing scheme.

This strength is best understood in the context of risk assessment — analyzing the potential cost of asset compromise and the cost to protect against that compromise, keeping in mind that protections in man-at-the-end scenarios generally ultimately fail when attackers have sufficient time and resources [66]. Where a risk of compromise exists, obfuscation can mitigate that risk to an extent, but at a cost. While different applications have different resource cost and performance overhead tolerance, establishing absolute obfuscation strength enables software providers to select appropriate transforms that provide maximal security within those cost constraints.

Absolute strength is multidimensional: it requires consideration of factors such as computation resources, skill competencies, time taken, and tools required in reverse

⁵Other work already provides significant insight into different obfuscation costs, such as energy use [274, 302].

engineering the obfuscation. Analyzing obfuscations to determine what it takes to defeat them is a difficult problem. Other work attempts to evaluate them in four main ways:

1. Provable security
2. Attack-defense scenarios
3. Software metrics
4. Human evaluation

I detail these methods next.

2.4.1 Provable Security

In the field of cyber security, different tools and schemes generally attempt to guarantee confidentiality, integrity, and/or availability of assets⁶ through design [279]. “Provable” security attempts to frame methods to protect assets in formal theorems in order to prove or disprove the protections’ efficacy [185]. This works relatively well in examining the hardness of very specific aspects of security systems; the Rivest–Shamir–Adleman (RSA) problem provides a famous example of this wherein the heart of the scheme relies on the difficulty of prime factorization to ensure confidentiality for the namesake cryptosystem [50]. In that example, decades of examining the problem have not yielded a proof that it is computationally difficult to break, but neither have mathematicians demonstrated ways to solve it with feasible performance. At the same time, researchers and security professionals identified a

⁶“Assets” here refers to a more general definition than “assets” in man-at-the-end attack; here, assets might be pieces of information but could also mean particularly to physical systems and components thereof.

multitude of RSA implementation vulnerabilities which do not rely on solving the RSA problem [247].

These “provable” security concepts have been applied to code obfuscation. About 5 years after Collberg et al. [96] formalized the concept of code obfuscation, Barak et al. [30] produced one of the first formal normative analyses of obfuscation hardness. There and in subsequent work, researchers enumerate the characteristics of ideal obfuscation schemes and attempt to prove or disprove their existence, where proving a scheme provides some idea of “provable” cryptographic hardness.

2.4.1.1 Blackbox Obfuscation

In a “blackbox” scenario, individuals (reverse engineers in this case) have a system that takes input and provides output, but the individuals do not and cannot know anything about how the blackbox actually functions beyond that.⁷ Barak et al.’s analysis focused on whether obfuscation can transform any program into a blackbox version of itself [31]. Their proof demonstrated that the combination of oracle access to program input/output and an obfuscated version of the same program resulted in a knowledge leak in a class of circuits/programs. Subsequent work came to the same conclusion when adding quantum elements to traditional computing primitives [10]. Other work focused on which types of circuits/programs could and could not be obfuscated with blackbox obfuscation, with significant results falling in both categories [133, 221, 116, 67, 336].

Just as with the RSA cryptosystem,⁸ the narrow focus and definitions do not cap-

⁷With this requirement, it is possible to embed a key into a program that reverse engineers cannot extract as an asset.

⁸The RSA comparison here is particularly poignant: As noted by Barak et al., Diffie and Hellman anticipated that blackbox obfuscation may enable implementations of public key cryptography before the RSA scheme (or any other public key scheme for that matter) had been developed [115]. As such, blackbox cryptography, should it exist in a strong form, could provide an alternative to RSA and other modern public key cryptographic schemes trivially by obfuscating an embedded key

ture practical implications of the concepts they analyze. That is, focusing solely on proving the difficulty of the RSA problem did not eliminate the many other vulnerabilities in cryptosystem implementations based on the RSA algorithm; the blackbox obfuscation analysis likewise falls short at their formulation of the problem. The proof demonstrates that some set of programs cannot be obfuscated without leaking at least some information about the original program, and that there at least exists some not-necessarily-easily-discoverable-but-polytime-performance algorithm to translate an obfuscated program to its initial state. However, this borders on practical irrelevancy: More than a decade after the initial proof, Barak reviewed developments since the initial proof that perfect and general blackbox security and found many applications of obfuscation through a less strict formulation called *indistinguishability obfuscation* which had been proposed at the time of the blackbox obfuscation analysis [29].

2.4.1.2 Indistinguishability Obfuscation

Shortly after Barak et al.'s initial blackbox proof, Goldwasser and Rothblum theorized that a significant amount of functionality provided by the non-existent perfect and universal blackbox obfuscation could still be implemented with *best-possible obfuscation* that leaked the least amount of information possible. However, this work noted that, at the time, only incredibly inefficient indistinguishability obfuscation — a category which best-possible obfuscators belong to — existed [134]. A decade later, work proposed and implemented significantly more efficient indistinguishability obfuscation schemes for evaluation. Garg et al. provide a formal definition of indistinguishability obfuscation before presenting their implementation thereof: [130]

Indistinguishability obfuscation requires that given any two equivalent
symmetric cipher function.

circuits C_0 and C_1 of similar size, the obfuscations of C_0 and C_1 should be computationally indistinguishable.

Computational indistinguishability requires that no efficient (at least polynomially performant) algorithm be able to determine the difference between a set of programs. As applied here, it requires that no algorithm be able to determine the totality of differences between two obfuscated programs. This would, theoretically, prevent an automated analysis from deobfuscating an obfuscated program P' back to the source program P , since no algorithm could find the differences between them to begin with.

By using multilinear jigsaw puzzles with homomorphic encryption, Garg et al. demonstrated that all circuits/programs can be indistinguishably obfuscated. However, related and subsequent work found this scheme (and other proposed indistinguishability schemes) to be reliant on unproven — and occasionally disproven — cryptographic concepts. Nonetheless, Jain et al. [168] demonstrate conclusively that indistinguishability obfuscation can indeed be universally applied. Currently, the proposed indistinguishability obfuscation methods incur massive performance penalties. Current work on the involved concepts such as homomorphic cryptography aims to make approaches like this more computationally feasible, and recent years have seen significant progress towards that end [322].

2.4.1.3 Relation to Practical Obfuscation Deployments

As discussed, blackbox — and later indistinguishability — obfuscation had been proposed as method to accomplish public key cryptography before the existance of schemes like RSA. In many ways, obfuscation parallels other cryptographic primitives in the potential to rely on sound, logical, provable security concepts to gain an idea of hardness in some contexts [150]. As also discussed, even this does not negate the

potential for vulnerabilities to exist — as they have in RSA implementations; this potential is a well-known problem with provable security wherein provably secure components do not necessarily result in secure systems or implementations — as in the case of side channel attacks [314].

Beyond these superficial similarities, however, lies a deceptively simple truth: Obfuscation strength is not the measure of how well algorithms can analyze programs after the application thereof. Rather, **obfuscation strength is a measure of what it prevents humans from doing with an obfuscated program.** Where cipher scheme analysis asks a simple question with a boolean answer — can this ciphertext be easily deciphered without a key? — obfuscation schemes ask something much more nuanced, nebulous, volatile, and irregular: Can a human accomplish their goals with this obfuscated program? Humans do not solely operate on orderly principles of logic and reason but rather introduce difficult-to-quantify judgment and instinct. As such, normative evaluation methods like those discussed here provide positive and negative guarantees of specific code obfuscation properties, but those properties are not necessarily relevant to reverse engineering difficulty. This illustrates another well known issue with provable security: Often, proofs often do not or cannot translate real-world security inquiries to formal problems well [185, 148, 184].

Blackbox obfuscation requires that a program not expose any of its internal operating information. In practice, only spilled knowledge which aids a human in reverse engineering is relevant, so blackbox obfuscation is not needed to secure programs; nor is blackbox obfuscation even sufficient in protecting against general reverse engineering — the input-output behavior alone with no knowledge of the underlying implementation of an obfuscated function can in fact be enough for a reverse engineer to accomplish their goals in some cases⁹ [53]. Similarly, indistinguishability

⁹This is the case when an attacker attempts to crack code protections or deobfuscate a learnable

obfuscation in itself does not imply strong protection against human attacks — if the knowledge spilled by an implementation of this type of scheme aids a reverse engineer, then it fails to the extent the human reverse engineer can use that to accomplish their goals, which is not necessarily to automatically translate the program to its original form. Thus, a proof of *unconditional security* for obfuscation requires assumption of an unbounded attacker and has not yet been shown possible (or impossible) with regard to software obfuscation, though it has been considered in the context of endpoint software security generally with other methods like hardware tokens [138].

2.4.2 Attack-Defense Scenarios

Provable security focuses on normative analysis regarding theoretical limits of obfuscation. Another type of normative analysis, attack-defense or red-team/blue-team scenarios, focuses on matching particular reverse engineering strategies with code obfuscations presented in prior work. This type of analysis seeks to show what reverse engineering strategies are known to work against particular obfuscations, and which obfuscations defeat those strategies.

Individual papers documenting obfuscations and reverse engineering strategies (see Sections 2.1 and 2.3) typically enumerate how effective the authors believe them to be in particular scenarios, with given goals and opposing strategies. Surveys curate a multitude of this prior research in order to generate a more general understanding of strong obfuscations are in light of known attacks [284, 5]. These surveys can be general, or focus on particular platforms or categories, such as Android or malware analysis [361, 293, 2].

This body of work provides valuable insight into current practices. It is, however [154].

ever, limited by the underlying analysis of techniques presented in individual papers. The hardness demonstrations of the obfuscation and reverse engineering practices therein tend to fall into one of the hardness analyses enumerated here — such as using provable security analysis (see Section 2.4.1) or generating metrics of an obfuscation (see Section 2.4.3) — and thus suffer the drawbacks of the type of analysis used. Occasionally, research uses human orchestrated attack and defense simulations (typically at a small scale, perhaps with just the authors) in order to analyze their contributions’ efficacy, which falls into human study analysis (see Section 2.4.4).

2.4.3 Software Metrics

Other work seeks to demonstrate obfuscation security by generating metrics from obfuscated code. Those metrics, then, determine the protection level of the software. Anckaert et al. [17] propose a benchmark suite with several runtime (dynamic) collected metrics, including:

- Instruction counts
- Cyclomatic numbers
- Knot counts

Cyclomatic numbers measure the number of decision points a program reaches, while knot counts measure the number of crossing edges in a vertical layout representation of a control flow graph — an assessment of control flow tracing difficulty. The authors theorize that these metrics measure the difficulty attackers have in reverse engineering programs generally. They collected these three metrics on non-obfuscated programs, obfuscated programs, and automatically deobfuscated programs with several transforms and were able to show significant differences between the program

categories in those metrics. In similar fashion but with fewer metrics, Talukder et al. [305] measure how well program slicing reduces dynamic program size. While the document lacks details, Sistany [107] proposes using heuristics to generate metrics for machine learning obfuscation hardness classification in a report curated from seminar discussions.

Work directly preceding this dissertation used categorical dynamic instruction counts in an attempt to capture program execution size and control flow complexity, all in order to determine if a program is or is not obfuscated compared to the original [309]. Kanzaki et al. [174] similarly use an N-gram model to differentiate between obfuscated and non-obfuscated code within a program. These methods can potentially be used to evaluate obfuscation strength, as they identify quantitative metrics of obfuscation — in their presented contexts, however, researchers only use them for categorical differentiation.

Using software metrics to evaluate obfuscation, however, necessarily assumes that those metrics demonstrate obfuscation hardness. This runs into many issues similar to normative evaluation: It assumes an attacker constrained by known reverse engineering methods that the metrics seek to measure. Without direct observation of these metrics making reverse engineering more difficult, however, they cannot reliably generalize to different obfuscation and reverse engineering practices [303].

2.4.4 Human Studies

As stated above: **Obfuscation strength is a measure of what it prevents humans from doing with an obfuscated program.** Until such time as a reliable model of the worst-case (and average-case, for that matter) human attacker can be generated for analysis, this definition requires studying human reverse engineers in order to determine what they can do and how well obfuscation prevents those activ-

ties in order to generate baseline, gold-standard practical obfuscation strength [75].

2.4.4.1 Psychological and Neurological Analysis

One line of research studies the psychological processes involved in reverse engineering [203, 59]. Wiesen et al. [341] conducted human studies of hardware reverse engineers solving synthetic problems with emphasis on these psychological principles. This follows similar efforts in software engineering, though the hardware reverse engineering study delves further into psychological aspects than current software studies.

Yu et al. approach this problem space from a neurological perspective, studying how human brains represent data structures with fMRI and fNIRS imaging; this impacts psychological understandings of reverse engineering processes, as reverse engineers often have to consider data structures as they work and these scans (and self reported thinking processes) provide insight into how individuals think to this [157].

2.4.4.2 Closed Studies

While a significant amount of software reverse engineering research contains some human-based data as authors simulate attacks and defenses on their own contributions, researchers have thus far attempted few open studies [349]. Sutherland et al. [303] timed 10 students who completed reverse engineering tasks and found some correlation between skill level and success but did not find software metrics to be representative of task complexity. Ceccato et al. [77, 76] contributed work studying Masters and PhD students attempting to reverse engineer identifier renaming, and later expanded that work by adding more students and transforms to their experiments. Hanash et al. [143] replicated this effort and determined that the obfuscations tested eliminated advantages based on subjects' programming experience. Viticchié et al. [325] continued this line of inquiry with an additional student-based study. In each case, researchers generated synthetic reverse engineering problems and gener-

ated challenges with original and obfuscated versions of those problems. Students reverse engineered both sets and filled out questionnaires regarding how they solved the problems. Differences between obfuscated and original problems demonstrated the difficulty of the obfuscation: the authors found that obfuscated problems required more time to reverse engineer.

Despite significant challenges in involving professional reverse engineers, other work attempts to examine them, generally with less detailed and quantitative data; Bryant [60] contributes an early case study interviewing professional reverse engineers and finding their practices consistent with the author's theory regarding reverse engineering processes. Ceccato et al. [79] followed up their research with studies using professionals rather than students. This later study qualitatively analyzed the professional reverse engineering subjects, in contrast to the more quantitative student based studies prior. This enabled deeper analysis of subjects' methods but generated less empirically observed, generalizable data. Ultimately, the authors distilled the qualitative data into impressively detailed attack models. Votipka et al. [327] follow with a similar professional interview methodology, distilling a reverse engineering model based on a sample of 16 professional reverse engineers (out of 68 initial participants) who met scheduling and survey response criteria; this work generated a model focused on three reverse engineering phases:

1. Overview
2. Sub-component scanning
3. Focused experimentation

2.4.4.3 Open Competitions

The efforts discussed so far use closed, localized environments in conducting human studies, constraining the subject sample stratification. Other work leverages open-ended environments to conduct human studies. Open challenges in computer security¹⁰ (and reverse engineering specifically) largely follow the model pioneered by the RSA Factoring Challenge, first introduced in 1991 [121]. In that challenge, subjects (individuals or, more often, sophisticated teams) attempted to factor large semiprimes; the RSA algorithm relies on the hardness of this factoring problem for cryptographic strength. While normative evaluation reduced RSA to this problem (though, as mentioned in Section 2.4.1, side channel attacks against RSA still exist) it could not prove the difficulty (or weakness) thereof [248].

In the absence of formal normative proof of RSA’s strength, RSA Laboratories (the company behind the cryptosystem) issued this RSA Factoring Challenge. This event published a series of large semiprime numbers and offered significant cash prizes (up to \$200,000 to subjects who factored them successfully up until 2007, when RSA Laboratories ended the cash prizes and declared the project inactive [176, 337, 9]). This, then, aimed to demonstrate the practical security of RSA — that is, the practical security of hard problem powering the algorithm protecting (at least at one point) the majority of encrypted web application traffic. Given the scale, expected difficulty, and importance of the challenge, authors who successfully factored larger semiprimes (on the order of 512 bits in 1999, 768 bits in 2010, and 829 bits in 2020) published their methods in detail [74, 183, 52].

Similar efforts RSA Laboratories put forth concerning the Digital Encryption

¹⁰Other computer science and mathematics subfields also use similar open challenge events to solve difficult problems, such as CoNLL shared tasks in natural language processing [357], ImageNet challenges in image detection [268], and the Millennium Prize Problems in mathematics, which notably includes an entry on P vs NP [71].

Standard (DES) cipher, where subjects earned cash prizes if they cracked DES keys in a short enough time period, successfully demonstrated its weakness [319]. Certicom concurrently offered the ECC Challenge dealing with elliptical curve cryptography, which demonstrated ECC key compromises up to 130 bits [24, 163, 41]. Several other challenges deal with less mainstream or standardized cryptography and security; these have demonstrated expected real-world hardness of those cryptographic schemes to varying degrees according to participation levels and problem framing [61]. The Cyber Grand Challenge studies how well different attacks might be automated through competition [56].

It is unclear the motivations behind subjects participating in these challenges — they may be solving problems for prizes, though in many cases (such as RSA Factoring after 2007) such prizes do not exist; they are likely solving challenges in the course of their own research and/or for notoriety, though it is not clear how often subjects publish their results in these challenges. In no known cases have challenge creators compelled method disclosure beyond solutions in order to participate. Similar efforts with similar structure have been used in conjunction with software protection methods.

The WhibOx series of competitions aimed to test the hardness of contemporary whitebox security implementations by offering challenges to the public at large. The results of these competitions conclusively demonstrated the weakness of the 94 tested implementations, as they were all defeated by subjects, most within the first day of competition [48]. This effort did not have official data collection channels beyond their submission (an extracted key) evaluation — subjects reported their methods in their own subsequent work rather than through the competitions themselves [136, 135]. The competition included cash prizes for both top contributors in challenge generation and solution submission categories, totaling \$800 and \$1200, respectively

in the latest round [332].

Ceccato et al. [80] also developed a public-facing challenge to test the ASPIRE framework. While this challenge received less participation than the WhibOx challenges with only one successful subject solving 5 of the 8 challenges, it exceeded WhibOx in its data collection: Using email follow-ups, the ASPIRE team determined the methods the subject used to defeat the challenges and applied this knowledge to the attack modeling framework developed with their professional reverse engineer studies [79]. Ultimately, the analysis added 23 new concepts to their taxonomy, bringing the total concept count to 169. 70 of the concepts previously observed in professional reverse engineer studies were also observed in this public challenge.

2.4.4.4 The Next Step for Human Studies

Human studies provide unique insight into how reverse engineers operate in practice — their methods, the efficacy of those methods, and the ability of obfuscation to stop them. Already, these studies have revealed a significant amount of data regarding reverse engineering methodology through closed studies and the state-of-the-art limits for those methods and accompanying software protection techniques through open competitions. The work presented in this dissertation combines these two methods: By using and expanding data collection methods similar to those in closed studies, this research seeks to gain insight into methodology details and generate analyzable data; by combining this with open studies, the data will ideally reflect the state of the art by removing closed study constraints. [307] Doing all of this requires significant expansion on current human behavior analysis methods.

2.5 Cyber Attack Models

Although prior work attempts to evaluate code protection in domain-specific ways, a significant amount of more general work in the cybersecurity field attempts to model attacks (such as reverse engineering) and defense (such as code protection). This dissertation expands these efforts in an attempt to build models from reverse engineering data, ultimately using those models to evaluate code protection techniques and reverse engineering practices.

In the cybersecurity (and related physical security) field(s), researchers and practitioners often use attack models in order to expose the vulnerabilities in systems and components. These attack models allow individuals to effectively communicate how attackers operate, including their attack vectors, resources required, and potentially even countermeasures available — a category of information known as *threat intelligence* [8, 187, 33, 328]. Attack models might be general, broad, and abstracted across different systems and components to describe general attack patterns, as in the case of the *Cyber Kill Chain* which models unconstrained attacks on very large, complicated systems without modeling the details of specific components of those systems [350, 23]. The *Diamond Model* provides a similar level of abstraction [65]. These models often focus on the motivations and behavior of attackers (using broad strokes) rather than the specific technical aspects of their actions [244, 229].

Other modeling methods might be more specific, modeling attack vectors for a constrained problem or system. This latter type of model applies to the work here, as it focuses on specific reverse engineering attacks on obfuscated binaries — a highly constrained problem. Examples of specific attack modeling include *attack trees*, *attack graphs*, and *Petri nets* [101].

2.5.1 Attack Trees

Attack trees model vulnerabilities by encoding desired states to nodes and placing requisite states to reach desired states as children. That is, for a given goal/output state, an attacker must first satisfy and achieve the children states [113, 160]. Children, in turn, may have “AND” or “OR” relationships — to achieve a parent state, either all of the “AND” children or one of the “OR” children must be achieved first. A variety of attack tree (as well as related fault tree and similar concepts) types exist, varying in implementation details that cater to specific needs [101, 153, 197]. For example, Pietre et al. [255] introduces trees based on Boolean logic driven Markov processes in order to generate dynamic and probabilistic aspects lacking in traditional attack trees. Likewise, attack tree generation varies in form — from manually generated trees based on individual attack case studies to semi-automated and automated analysis methods based on available synthetic and real-world data sources¹¹ [22, 170, 323, 241, 64]. Prior work demonstrates attack trees built with varying levels of specificity and generalizability¹² for attacks compromising medical implants [292], databases [331], CubeSats (small satellites) [122], supervisory control and data acquisition (SCADA, often used in industrial and critical infrastructure) systems [311], vehicles [177], IOT devices [64], smartphone applications [363], utility smart meters [231], and network security [114]. Some work generates attack trees for particular, specific software and hardware reverse engineering as well [335, 254]. Attack trees, then, experience widespread use in the cyber security field.

¹¹Data sources used to generate attack trees tend to be highly domain specific, such as iterating through and selectively and theoretically invalidating system policies to generate insider threat attack trees or using known vulnerability databases combined with network configurations to generate potential attacks on that system [165, 46].

¹²The data gathering methods employed in individual works influence their general applicability; most works conduct case studies on a limited number of examples.

2.5.2 Attack Graphs

Similar to attack trees, attack graphs represent steps in a cyber attack as nodes. Unlike an attack tree, these nodes have no hierarchy, instead consisting of initial state (source) nodes, outcome state (sink) nodes, and intermediate nodes [196]. The edges between these state nodes correspond to an actor doing something to further the attack. A completed attack consists of a traversal from an initial state to an outcome state; this contrasts to attack trees, where an attack consists of a traversal from leaf node(s) to the root node. This difference allows attack graphs to better model complicated attacks, as steps can be considered in all contexts during complicated traversals rather than just in the context of one tree branch [101]. However, attack graphs tend to be more difficult to analyze in static contexts due to path explosion when considering many different potential attack vectors [197, 243]. As a result, a significant amount of effort attempts to reduce attack graph visual complexity [152, 243] or convert between the two forms [144, 283].

Just as attack trees can vary in implementation details for specific purposes, some areas of research enhance attack graphs in order to better attack particular problem domains. Of particular importance, one active research area enhances traditional attack graphs with probabilistic edges in order to build Bayesian attack networks [127]. This enhancement allows practitioners to better predict and follow attacks, even in real time [346].

As with attack trees, many methods attempt to generate attack graphs with manual, automated, or semi-automated methods with data sources (scanned network hosts or intrusion alerts, for example) similar to those found in attack tree generation [179, 246, 161, 180, 180, 236]. Attack graphs also mirror attack trees in their industry acceptance and popularity, with notable work producing attack

graphs for security aspects of microservice architectures [159], network security [161], medical devices [218], ransomware [368], digital forensics countermeasures [82], and vehicles [275].

2.5.3 Petri Nets

At their core, Petri nets are directed, bipartite graphs consisting of a set of places, a set of transitions, and a set of arcs/edges; places correspond to a given state, and transitions correspond to actions [43]. Edges connect each place to at least one transition, and at each transition to at least 1 place. Places contain zero or more tokens, and transitions require their source places to have a certain number of tokens in order to fire/execute. When transitions fire, they add token(s) to their output place(s), as shown in Figure 2.6. These tools have long been used to model complicated, concurrent systems by keeping track of what results from actions (the tokens in places) and what happens with those results (what can be done with tokens in a given place) [42].

Though perhaps not as popular as either attack trees or graphs, cyber security researchers and practitioners employ them to model attacks [342, 334, 215, 19]. Similar to attack graphs, an attack on a particular entity consists of a traversal through the Petri net’s graph. Petri nets demonstrate superiority, however, in modeling asynchronous and distributed attacks because it can contain multiple states for multiple entities; they are sometimes used in conjunction with the other attack models in order to take advantage of this feature [101, 195, 338, 317].

As with attack trees and graphs, Petri net enhancements and variants exist. In particular, generalized stochastic Petri nets introduce probabilistic firing schemes [90, 227] which enables further predictive analysis by suggesting what an attacker is most likely to do [358, 101, 12].

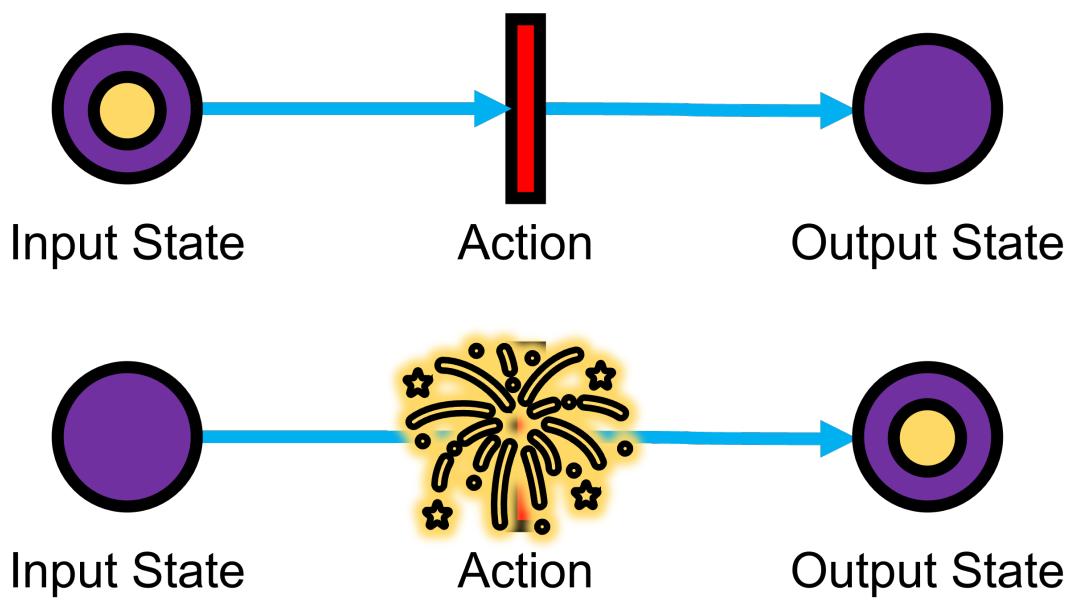


Figure 2.6: A basic, single action Petri net in its initial state and after an actor “fires” its action, moving the token for the actor from the input to the output state. Transitions are shown in purple, transitions in red, and tokens in yellow.

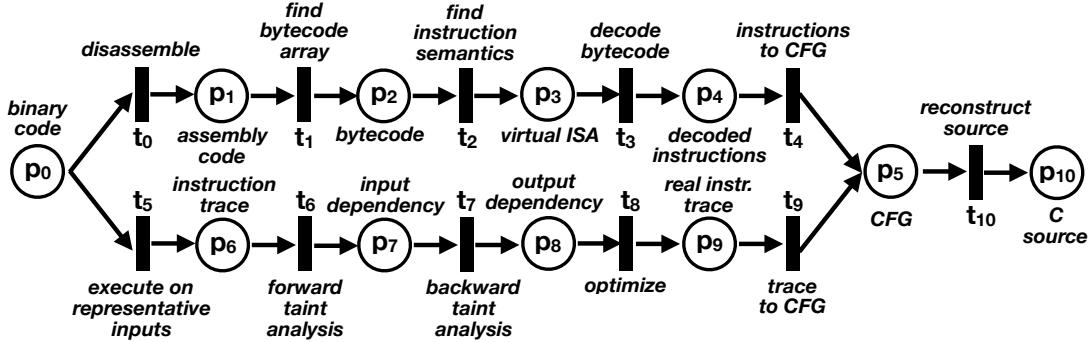


Figure 2.7: An example Petri net proposed (and reproduced from) prior work on RevEngE. [307] This Petri net does not reflect real-world observations, but instead serves as an example and output goal for reverse engineering studies.

To date, most Petri net generation work uses manual case study analysis done by experts, as done by Nasr et al. [239] in generating attack models for SCADA systems or Li and Li [207] for traditional network security. Prior work on RevEngE [307] provides an exception to the manual Petri net modeling. That work proposes (but does not implement) Petri net models for reverse engineering built on data collected from expert reverse engineers; it provides an example of expected output as shown in Figure 2.7. In that example, a reverse engineer attempts to deobfuscate and decompile an obfuscated binary. To do so, they take one of the two paths shown; each circle is a state corresponding to a step in the reverse engineering — such as the starting state with binary code and the ending state with C source — and each vertical rectangle is a transition representing an action to get from one state to another — such as disassembling the binary or doing taint analysis. A reverse engineer undertakes a series of actions, each action transitioning from one state to the next, to get from the initial compiled binary state to the successful C source state. That series of actions is a path in the Petri net.

Petri nets, in particular, fit this system well — places correspond to attacker states and transitions correspond to attacker actions. As will be detailed in Chapter 5, user trace data ideally captures users’ actions and provide an idea of attackers’ states — the data, for instance, can reveal when an attacker disassembles code or builds CFGs as shown in Figure 2.7’s example; this, in turn, enables semi-automated Petri net generation, thus accomplishing some of the goals described in the prior RevEngE work. Ultimately, the work presented here aims to build Petri nets which illustrate reverse engineering strategies.

2.6 Building Models from Human Behavior Observation

In building models of reverse engineering techniques, this dissertation hinges on effective approaches to (1) *observe* human behavior as experts reverse engineer, and (2) *analyze* those observations into useful models of reverse engineering methods. Several subfields of computer science — particularly human computer interaction, computer security research, software engineering, and pedagogical multimedia sharing — attempt to observe and model human behavior for other purposes. Elements from that research parallel and influence the work presented in this dissertation.

2.6.1 Observation Methods

Four areas that commonly monitor humans as they use devices, namely (1) human computer interaction research, (2) computer security, (3) software engineering research, and (4) pedagogical multimedia sharing, offer relevant approaches which can be leveraged in order to observe humans as they solve problems using their methods. Capturing these methods — illustrated in Figure 2.8 — in data enables the research here to examine how reverse engineers solve practical challenges. The first relevant area of research monitors users in order to determine how to gauge efficacy

of different interface methods; the second attempts to detect and prevent or counter attacks on systems; the third studies software developers' methods; the final records individuals attempting to share their methods with others as instruction.

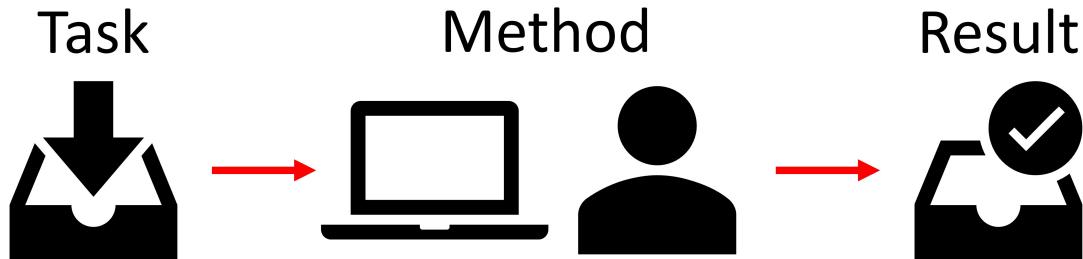


Figure 2.8: Individuals use methods to complete tasks on their devices. The research presented here attempts to capture the “Methods” used in reverse engineering.

2.6.1.1 Human Computer Interaction Research

Research on human-computer interfacing (HCI)¹³ necessarily involves examining individuals using devices. Examples include evaluating eye tracking input systems [191], assessing graph visualizations in domain specific settings [112], and provisioning event sensors within a program to detect what a user does in the context of quality assurance [270]. While the specific subject matter in this area varies to a large extent, methods of evaluation to date involve highly specific observation setups for experimentation: In the former example, individuals are observed with device cameras as they look at dots on those devices, while the latter observes subjects in person as they use the presented graph visualization.

¹³HCI expanded into broader Multimodal Human–Computer Interaction (MMHCI), a closer analog of HDI in scope — it includes devices besides personal computers — than traditional HCI [167].

2.6.1.2 Computer Security

In computer security, system administrators often monitor the resources on their networks in order to detect threatening behavior. This involves monitoring what computers do on the various nodes; tools like syslog-*ng* [264] and Splunk [70] offer monitoring of native resources, while other tools such as Snort [73] monitor network resources. Other related work proposes using other user-based behaviors such as login/logoff, file access, and email in order to classify user behavior as malicious or benign [355]. While these tools offer some desired functionality, such as program launches, in collecting data on users' methods in completing tasks, they do not deal with large amounts of user interface based data. This, in turn, limits their usefulness in monitoring user behavior by excluding what a user does inside of a launched program — including user input through the mouse and keyboard and the resulting program's behavior. All of this limits what data may reveal about subjects' methods, as the type of program run (which is what syslog and Splunk record) and network packets (as with Snort) offers limited information on what a user does as they interact with a device.

2.6.1.3 Software Engineering and Medical Imaging

One area of research in software engineering focuses on how engineers' methods — again, the “Method” shown in Figure 2.8. Initially, researchers used a fairly standard interview-based methodology to understand how software engineers conducted tasks such as thinking about data structures for particular applications [4, 253]. In some ways, this aligns with the small-scale and subject-matter limited human computer interaction research presented in Section 2.6.1.1 in that the methods limited the scope of the inquiry. Later work, however, studied software engineering in a more open environment while observing the interaction through neurological and

physiological methods, including fMRI, fNIRS, and eyetracking in addition to the self-reporting methods [287]. Using medical imaging shows promise in evaluating aspects of human-device interaction — the researchers found significantly different patterns in activity concerning prose and programming languages, for instance — but the area needs more work in order to be generalizable to more human-device interaction activities.

2.6.1.4 Multimedia Sharing

Over the last decade, screencasting technologies enabled individuals to share some aspects of HDI data (specifically screenshots) to viewers remotely, either in real time or after recording [216, 139, 222]. These technologies allow individuals to share a significant amount of their methods to others without constraint, but suffer several drawbacks concerning the research presented here. Current research in this area focuses on pedagogical uses for this technology which, at their core, involve communicating methods from instructors to students [339]. The data produced — raw screenshots with optional narrating audio in cases where the author knows in advance what exactly they are recording — lacks structure, which makes viewing or analyzing it in any format other than raw video difficult. While screenshots are used in the research here and contain a huge amount of data, analysis of just image data like this either relies on image analysis technologies (which have not been developed for this problem domain, yet) or a huge amount of human analysis that other types of data might streamline [256].

Prior work by Intharah et al. [162] supplements screencasting with other HDI data sources, including keyboard and mouse input, using disparate tooling in small-scale experiments typical of HCI research as discussed in Section 2.6.1.1.¹⁴ Using all of this

¹⁴Intharah et al. adds keystroke and mouse input data from the operating system, but also leverages other image analysis work to derive features such as UI widgets or on-screen text [345,

data and semi-automated, supervised analysis, the work attempts to distill human-understandable representations of the subjects' methods through *programming by demonstration* — though evaluative “methods” in this work is limited to relatively small tasks such as skipping ads in a YouTube video or creating a spreadsheet, and scalability to larger scale tasks like reverse engineering a program is unclear [162, 206]. The methods here expand on these efforts by integrating similar but expanded tooling into a unified architecture which can easily be applied to open environment, remote experiments. Additionally, the work here attempts to separate method analysis from subjects, so that subjects need not worry about modeling as they attempt to solve already-difficult challenges — a vastly different task than sharing known methods on known problems in an instruction-oriented setting as in prior work; as such, the work here generates a differing analysis pipeline.

2.6.1.5 Limits of Current Observation Methods

While human observation methods in computer security are broad in scope, they lack the depth that human interaction research methods offer, and the inverse is also true in that interaction research lacks broad scope. Medical imaging studies study individuals' methods by observing activity in the brain rather than on the device — a much more complicated and less structured system that requires more research before it can be leveraged in open studies; medical imaging also requires rare or expensive hardware. Finally, screencasting and multimedia sharing techniques do communicate a large amount of highly detailed methodology data, but the video analysis thereof requires large amounts of human analysis; current research efforts have also not explored using screencasting for crowdsourcing methods as done here with reverse engineering.

[366, 89]. Some of this data, however, can be directly obtained from the operating system, as is done here with windowing information.

To extract what methods individual users use while completing tasks in unconstrained environments, both breadth and depth are needed, as alluded to in work discussing specific requirements to determine individuals' methods of reverse engineering. [307] Moreover, the data types providing this breadth and depth must be analyzable; medical imaging suffers from limits of current data analysis methods — the expansion of which is not within the scope of this dissertation.¹⁵ Multimedia streaming demonstrates promise but benefits from supplemental data and requires adaption for the type of work here. These requirements are typical not just of reverse engineering, computer security methods, or of computer science generally — they are universal across people using devices without constraint: When there are no constraints on what an individual does with a device, monitoring the input and output HDI captures how an individual uses the device. In the research presented here, I determine if this can be translated into what an individual is doing with their device, the “Method” shown in Figure 2.8.

2.6.2 Analysis Methods

The goal of this research involves extracting human-understandable representations and analytics of methods from users as they complete reverse engineering tasks, specifically through remote HDI data collection and analysis. Human observation data is a generally¹⁶ time series. Time series data can be thought of as a set of data points with an observation and a timestamp: $x_1, x_2, x_3, \dots, x_t$ where x is a piece of data and t is a timestamp index.

¹⁵Medical imaging showing brain activity requires significant amounts of work in order to connect that data to actual methods used.

¹⁶Some pieces of data, such as the operating system or architecture of a device or demographics of an individual, may not be time series data and are also important, particularly in how they relate to time series data. Importantly, the time series data captures what individuals do as they use the device, where environmental variables provide context for the time series data.

In this dissertation, I examine a combination of HDI time series data sources not yet studied in prior work; the data is collected as users complete tasks. The analysis of this data focuses on deriving the high level tasks from the underlying data through an annotation process (as is common for some time series data analysis) and building models from the combination of low-level HDI and high-level task data. While some prior work demonstrates significant similarity — such as in clickflow (user click logs in specific application settings) analysis [273] — the work here adds several dimensions to the prior efforts dealing with HDI-type data. While other time series data research, such as those listed above, can have high dimensionality, to date no other work aims to analyze the specific HDI data types proposed herein. Moreover, the combination of data types — user input, window, process, and screenshot sources all containing combinations of numerical, categorical, and ordinal data — add complexity prior work generally does not confront. Nonetheless, the methods used in prior work inspire and inform the methods here — particularly in efforts seeking to visualize and annotate the data in order to build models from it. Due to the sheer amount of diversity in prior time series analysis methods, prior techniques used in combination mesh well with the diverse data HDI data sources recorded here.

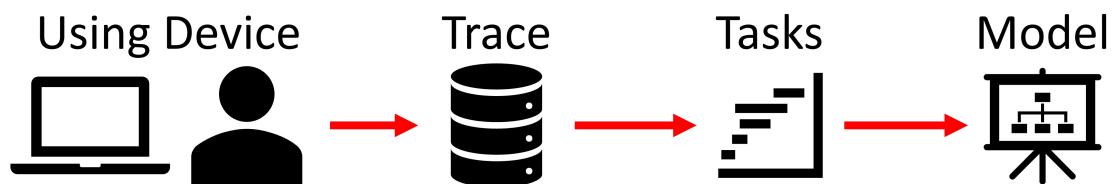


Figure 2.9: Individuals using devices generate time-series data as traces. Analysis methods focus on extracting tasks from these traces and then integrating tasks and raw data into models.

2.6.2.1 Modeling Process

Time series data does not often contain high-level, human understandable data on its own; rather, it generally contains granular, small scale data generated by narrow observations. The data must be analyzed and the high level understandings extracted through that analysis. Analysis methods range from using simple graphs, regressions, or summary data to make high-level conclusions (as is often done to show correlation models [291, 88]) to more sophisticated annotating/labeling underlying data for further analysis — as presented in somewhat analogous work on air quality sensor data [235] and audio files [69]. This is the case with the data collected here: Given the environmental constraints of the experiment, the method here assumes only low-level HDI data availability.

Following analysis, the high-level understandings can then be transformed into a useful model¹⁷; in the case of the reverse engineering task HDI data examined here, a useful model is defined as one which conveys the methods a subject uses in a human-understandable form. As detailed in Section 2.5, two computer security domain-specific modeling methods are *attack trees* and Petri nets [300, 359, 35, 330]. These types of models encode the methods an individual uses to achieve a goal, which aligns with the goals of the work here broadly in analyzing task completion and narrowly in reverse engineering subject matter and computer security narrowly. This research aims to present data in the latter form — Petri nets — which require a series of subject actions (steps in a process, the process here being reverse engineering for a challenge) as inputs. Those steps can be thought of as "tasks," the things a subject does while recording their HDI data.

¹⁷Model types vary significantly according to specific research purposes, but often range from the *Petri nets* of this research to classifiers built with machine learning techniques [175].

2.6.2.2 From Traces to Tasks

The first step in generating models involves analyzing traces in order to extract high-level, human-understandable tasks describing the methods employed by subjects. “Tasks” are labels of the underlying trace data; they describe what the human is doing in the trace. Time series annotation is a well known problem, though the high dimensionality and type of data generated here does provide some unique differences with previous work. Researchers often seek to annotate time series data: Individual energy use data can be annotated with appliances running, data flows can be annotated with program behaviors, astronomical data with observation labels, and so forth — many of the enumerated disciplines listed in Section 2.6.2 make use of annotation [40].

Time series data annotation is often employed in both supervised and unsupervised data analysis [299, 51, 285]. In supervised analysis, humans annotate data, sometimes using sophisticated tooling and visualization; in unsupervised analysis, algorithms (such as clustering algorithms) or models (sometimes built on previously annotated data or rule-based observations made by researchers) annotate data [280, 86]. Sometimes, semi-supervised annotation uses algorithms and models to aid humans in annotating data, significantly reducing the amount of manual annotation [286, 267].

Across disciplines with data for which no naturally applicable rules exist, supervised annotation provides the benefit of at least approximate *ground truth* — gold-standard data annotation generated by humans which can (ignoring biases and methodological concerns) be considered generally accurate and correct.¹⁸ [289, 72, 124, 132, 166] With complex human behavior based datasets, ground truth does not exist; complex human behavior and intuition do not generally follow readily apparent

¹⁸In practice, supervised annotations must be evaluated to determine their accuracy, generally by using more supervised, human analysis. Practices vary according to the actual data [295, 318].

models. In such cases, some level of human supervision encodes that human element into data. This is the case with the reverse engineering tasks studied here; it is also the case with the underlying HDI data, wherein humans do complex things with their devices. As such, the research here requires supervised annotation.

Li et al. [205] present the most analogous problem to HDI trace analysis appearing in prior work — screencast tutorial video analysis, which attempts to annotate videos without any other HDI data sources. Paralleling the reasoning here,¹⁹ that work adopts a low-level to high-level data analysis pipeline relying on human unsupervised (and later semi-supervised, for low confidence labels) annotation, generating high level understanding from low level screencast data. This work utilizes established methods to aid in the annotation, including building automated video segmentation (cutting the video into multiple parts) from screenshot cue classifiers, where significant changes in cues indicates a segment change. The researchers built the cue classifiers via a standard video classification supervised annotation model. Workers²⁰ then annotate the screencast segments with high-level labels via a web tool. All of this builds a usable model, which can be used to automatically annotate new data with machine learning processes; labels can then be used for purposes such as video search or captioning. This work recognizes the potential for future work to include more data sources (including, potentially, more HDI data) into the “cues” that they use to aid in segmentation:

In the future it may be illuminating to include more cues into the frame-

¹⁹Tutorial videos demonstrate human behavior, and thus generally have no ground truth for unsupervised analysis, making supervised annotation appropriate. Other work deals with user-supplied tutorial video narration for unsupervised analysis, as that narration itself contains ground truth [11]. The dataset collected here does not have any such ground truth data source.

²⁰This work limits itself to a small number of popular software domains, mainly Adobe Photoshop, and uses crowdsourcing platforms to find experts in those areas to use as workers. This departs from the work here, as there exist fewer reverse engineering experts to leverage as workers.

work, such as OCR as well as other non-visual signals e.g., speech-to-text transcripts, user logs, etc.

Many of the methods used in prior tutorial analysis may be applied to the datasets generated here — much of the work can potentially streamline the annotation pipeline here — though much is considered out of immediate scope and reserved for future work. Additionally, the dimensionality of the data here diverges from the plain video, adding windows information, process information, keystroke and mouse input; the labeling dimensionality likewise exceeds the previous work, as annotations for the models used here imply label hierarchies, as detailed in Section 5.4. Other work dealing with this level of dimensionality in data has even lower levels of dimensionality in annotations, such as classifying user behaviors as anomalous or benign [355, 3]. This work also uses a synthetic dataset — limiting its potential use as ground truth — with no need for supervised annotation.

Although no prior work demonstrates annotation methods dealing with the dimensionality required here, as mentioned piecemealed methods across time-series analysis prove valuable in aggregate for the work here. Timelines, Gantt charts, sequence charts, time-based data plots, summary data fields, video display, and filtering methods from this related work all find use for this research. Encoding the data sources in these visual representations while enabling concurrent annotation methods — such as those shown in some types of video annotation [142, 103, 57] or GPS movement on a timeline [141] — enable experts to annotate reverse engineering HDI traces in order to build models. Bellamy et al.’s CogTool, a programming task modeling tool, generates data remarkably similar to the HDI data recorded from users here — though with dissimilar, more limited program specific methods — and visualizes this data using many of these methods; their visualization similarly builds

on these time-series analysis methods and produces several closely related components [39]. That work, however, does not enable experts to closely analyze and annotate traces. Rather, CogTool assumes the operator knows exactly what they did to produce the trace.

2.6.2.3 Building Models

Once experts annotation low-level data to high-level task labels, further analysis translates those tasks into usable models — specifically, Petri nets. Petri nets consist of states and transitions as nodes in a graph structure; as pertains to security, states describe the status of an attack with respect to assets, while transitions describe attackers' actions [230]. Current work describes methods to visually encode different data attributes into these data structures, such as ways to differentiate between cyber and physical based attack steps, deal with abstraction level, or embed elements from other models [87, 364]. Other work focuses on ways to analyze these models, such as by optimizing costs associated with attacks, or builds Petri nets for specific problem domains [81, 146]. Still other work describes tools and methods to build Petri nets, but no prior work describes automated ways to do this based on other cyber attack data, HDI task traces or otherwise [260]; some prior work does demonstrate automated generation system design reliability diagrams, though that work approaches the problem with dissimilar, design based input data does not inform methods here [199]. Palanque et al. [249] demonstrate a manual method to translate user task data into Petri nets in the context of system interface and functionality design assurance; while not completely analogous to the work here — it uses a different modeling structure with manual (rather than data-driven) inputs and models and focuses on user device interaction rather than more open-ended attack vectors — the relationships between tasks directly correlate to the assumptions made in task mod-

eling and Petri net generation here. The work here extends cybersecurity based Petri net generation efforts by building Petri nets automatically from high level reverse engineering task data.

2.7 Summary

This dissertation aims to empirically build models of software reverse engineering practices in order to evaluate the efficacy of those practices associated code protection techniques. This, in turn, requires an understanding of current practices regarding man-at-the-end attacks (discussed in Section 2.3) as well as attack modeling (discussed in Section 2.5) and the human behavior observation and analysis methods needed to produce those models (discussed in Section 2.6). Specifically, this dissertation contributes to the field by observing reverse engineers in practice in order to build models of their behavior and expertise.

Chapter 3

THE CATALYST DATA COLLECTION ENGINE

The Catalyst Data Collection Engine attempts to meet the needs described in Section 1.1. It contains components which gather and archive data from individuals, visualize that data for experts to analyze, and build models based on the analysis. Links to the codebase for the Endpoint Monitor and backend architecture can be found in Table 1.2.

In this chapter, I will discuss the relevant device requirements and performance goals (3.1), the Catalyst architecture (3.2), the Endpoint Monitor (3.3), and the back end server (3.4).

3.1 Requirements, Goals, and Assumptions

Catalyst must record HDI data from subjects' devices, curate that data, and analyze that data to generate models. While Catalyst strives for maximal compatibility with different environments, it is not possible to support every single hardware and operating system configuration for every single device. This section provides an overview of the expected computing environments for different Catalyst components, and summarizes its requirements in Tables 3.1 and 3.2.

HDI data consists of all available user input and output interfaces on the general-use commodity hardware and software¹ subjects are assumed to use. This consists

¹General-use commodity hardware is currently assumed to be x86 or ARM devices and commodity software includes typical modern operating systems that run on these architectures, including Linux, Windows, and Mac. It is expected that commodity hardware be relatively modern, with at least 4 GB of memory. Table 3.1 lists other environment requirements; Catalyst was built with a Raspberry Pi 4 (RPI4) performance envelope in mind, as it is a relatively high powered yet inexpensive ARM device that could be used for reverse engineering ARM software [232]. Most x86

of keyboard and mouse input interfaces, as well as screenshot and active window output interfaces. Process information also records other important data, such as details regarding processes connected to windows, particularly when not in focus and active. These components are shown in Figure 3.1.

Catalyst presents two main use cases for subjects, depending on architecture: (1) a user has a reasonably powerful x86 device and runs Catalyst within a virtual environment, and (2) a user has a relatively lower power ARM device that runs Catalyst on the bare hardware. In either use case, users create a new environment specifically for a given event,² which potentially removes unrelated data (like background processes a user might be running) from the traces. In the former use case, virtualization allows users to shield their more sensitive HDI input and output data and operating system environment from Catalyst monitoring; this minimizes potential password and private data exposure to study organizers and — if published — the public at large. Virtualization also minimizes the risk of Catalyst side effects, such as filling storage or creating an unstable environment if a users' environment setup does not perform sufficiently. In the latter, such data would not be present by default, since users burn a new operating system for the event.

Subjects may be located anywhere in the world; Catalyst aims to maximally support subjects' environments. It is assumed, however, that those subjects can connect to the internet (or a specific network for closed deployments of Catalyst) with their devices and access web traffic to connect with a public server without firewalling issues and with reasonable modern bandwidth. Nonetheless, subjects' connections may experience periods of lossiness and collection of data cannot rely on

devices exceed RPI4 performance, and virtualization performance penalties will not likely eliminate this advantage [131].

²Users configure a new virtual machine for the former case and burn a new operating system to persistence like an SD card for the latter.

an active connection, as long as users later connect with the device.

Requirement Type	Minimum
Operating System	Windows or Linux
Architecture	x86/64 or ARM 32/64
CPU Cores	2
Memory	4 GB
Internet Bandwidth	1 MB/s

Table 3.1: Catalyst Endpoint Monitor minimum requirements.

Requirement Type	Minimum
Operating System	Linux
Architecture	x86_64
CPU Cores	4
Memory	16 GB
Internet Bandwidth	>100 MB/s

Table 3.2: Catalyst Server minimum requirements.

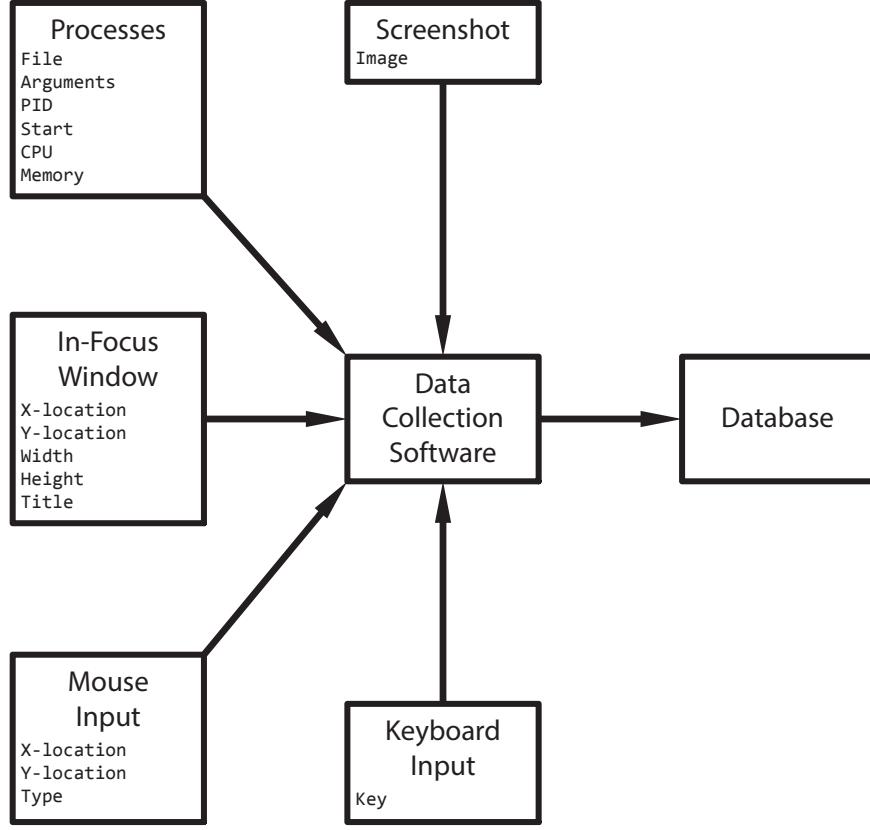
It is assumed that Catalyst will have an available web server with a DNS entry and fast connection to the internet at large, maximizing its reach and availability to subjects around the globe. That web server is expected to have a sufficient storage commensurate to the number of subjects and the length of their time spent, and at least 16 GB of memory as well as several processor cores available. For the experiments here (the various types of RevEngE and The Grand Re, which expect

on the order of 10s to 100 of participants) the research team used a 2014 Mac Mini matching this description. Though this hardware proved sufficient for the enumerated experiments, the Catalyst system should be able to scale to larger experiments with more performant hardware.

Catalyst aims to be simple, secure, and performant for subjects to use, creating as little overhead as possible while running. It also aims to support as many typical computing environments used by subjects as possible, and to be easily deployable on servers with different computing environments as well.

3.2 Architecture

The Catalyst Data Collection framework consists of two main software components shown in Figure 3.2. The web server manages authentication, provisions installers for the Endpoint Monitors as shown in Figure 3.3, collects and stores data from the Endpoint Monitors, does create, read, update and delete (CRUD) operations on that data, and contains the pages with the visualization and analysis tools as shown in Figure 5.3.



$$D_{total} = \left\{ \begin{array}{l} D_{window} = \{ w_1, w_2, w_3, \dots, w_x \} \\ D_{process} = \{ p_1, p_2, p_3, \dots, p_x \} \\ D_{keyboard} = \{ k_1, k_2, k_3, \dots, k_x \} \\ D_{mouse} = \{ m_1, m_2, m_3, \dots, m_x \} \\ D_{screenshot} = \{ s_1, s_2, s_3, \dots, s_x \} \\ D_{annotation} = \{ a_1, a_2, a_3, \dots, a_x \} \end{array} \right\} \mid n_x[Time] < n_{x+1}[Time]$$

Figure 3.1: Catalyst collects several types of data while subjects use their devices. A more precise explanation of the data sources and fields collected for them will be presented in 3.3 and 9.6 respectively. The exact structure of this data in the database, including implementation details omitted here, can be viewed in Appendix E's entity-relationship (ER) diagrams. For later consideration in visualization and analysis algorithms, D_{total} is the set of sets of datapoints — it contains the sorted lists of all datapoints from all data types.

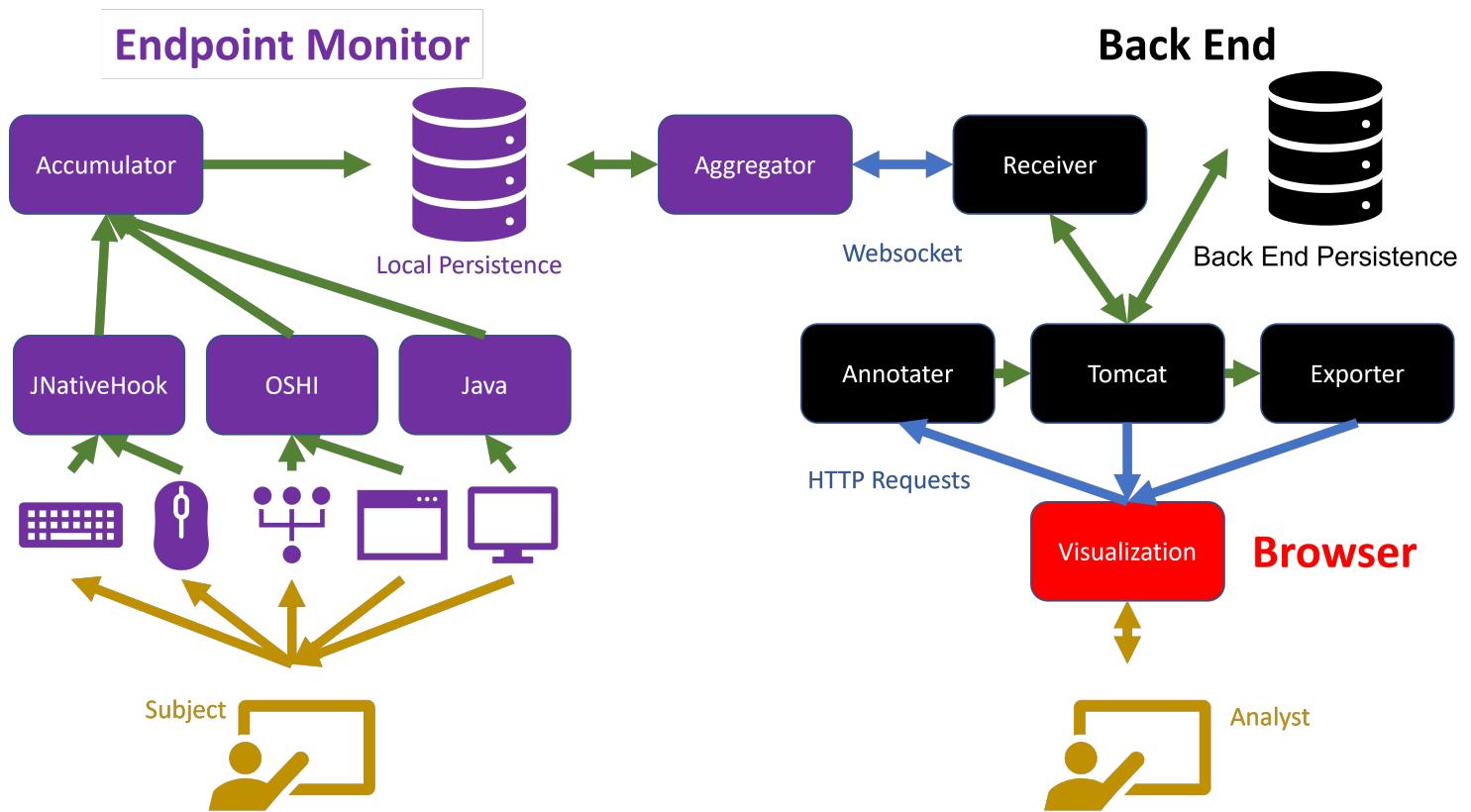


Figure 3.2: The main components of Catalyst include the Endpoint Monitor in purple, the Back End in black, and the Front End in red. Humans are colored yellow, local data flows green, and web data flows blue. Note that currently Catalyst uses local databases for persistence, but this could be reconfigured as remote. These implement the fundamental of Catalyst — capturing HDI data (of the type illustrated in 3.1) and presenting it to analysts in useful ways which often involve annotating that data.

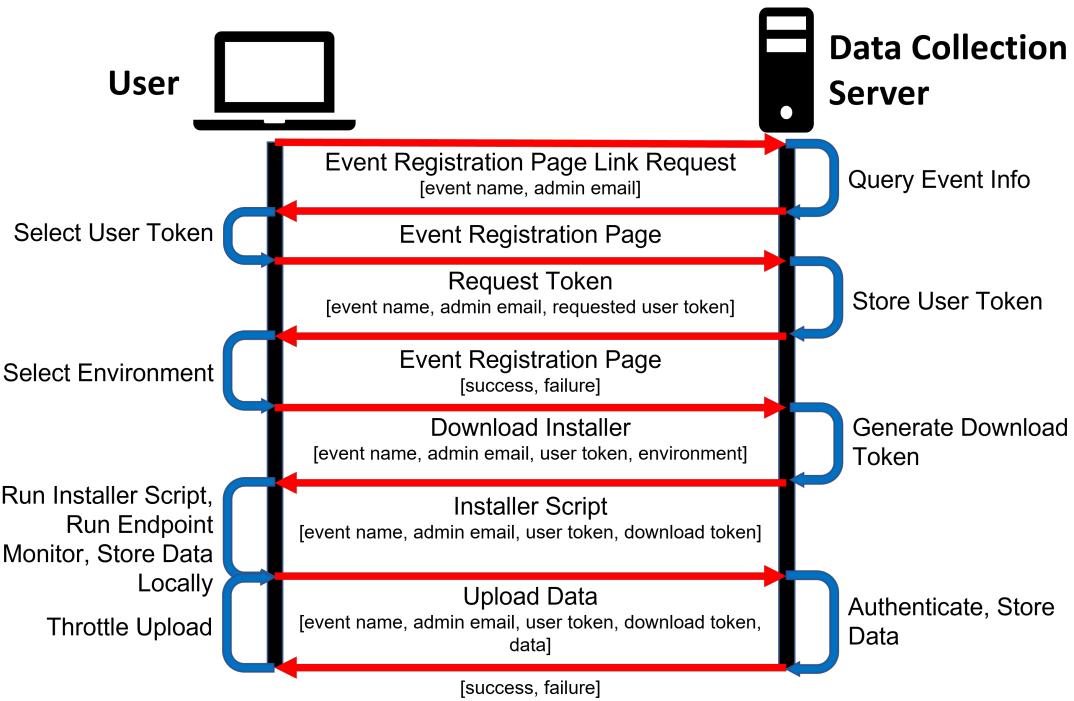


Figure 3.3: To use Catalyst, users sign up via a web application, download and run an installer, and run the endpoint, which continually uploads data that it collects from users’ devices. The red arrows indicate requests/messages and responses using standard protocols such as the hypertext transfer protocol (HTTP) or its websocket (WS) extension, while blue arrows indicate processes running on client or server devices.

3.3 The Endpoint Monitor

The Endpoint Monitor runs on users’ devices in order to collect and store HDI data before sending that data to the back end server for curation. As shown in Figure 3.3, the provisioning and upload process follows a typical web architecture — subjects first send registration information to the server and authenticate using that registration in their web browsers. That authentication then provisions an installer which subjects download and run to install the Endpoint Monitor, and then

the device runs the Endpoint Monitor. The Endpoint Monitor uses the provisioned authentication information to upload collected data to the server via a standard WS connection.

Using web-based tools for these steps maximizes compatibility. However, this process presents two main technical challenges:

- Sourcing the data in heterogenous computing environments;
- Sending large amounts of data from geographically distant locations with potentially lossy connections to the back end server, as detailed in 3.4.2.

3.3.1 Sourcing the Data in Heterogenous Computing Environments

Given that users may utilize many different environments and tools when interacting with devices to execute tasks, the Endpoint Monitor must be portable in order to accommodate those environments and tools. The Endpoint Monitor must also be performant, so as to not impact users as they work on solving challenges.

The types of data needed — keystrokes, mouse input, active windows, processes, screenshots — are typically managed by operating systems and associated components. Competing operating systems often differ significantly in behavior regarding functionality related to these data sources. Even different versions of the same operating system can have much different implementations of the same functionality, such as when Ubuntu moved from the classic X windowing system [282] to Wayland [301]. Nonetheless, some functionality tends to be uniform in implementation, such as POSIX standards. [21]

In order to build a maximally compatible and system while maintaining a high level of performance, the Endpoint Monitor is implemented in Java. The Endpoint

Monitor generates screenshots via the standard Java API’s Robot class,³ captures keyboard and mouse input with the JNativeHook library,⁴, and uses the OSHI library to capture process and window data.⁵ All of these libraries support common architectures and operating systems, including Windows, Mac, and Linux running on x86 and ARM variants; data is captured on an interrupt basis where possible — such as upon user input — and polled where it is not — as is the case with process and screenshot data.

3.3.2 Storing the Data

Because subjects using Catalyst may have intermittent internet access, the Endpoint Monitor must first store data in local persistence before uploading it or otherwise risk data loss. Whenever the Endpoint Monitor detects a connection, it streams data from the local persistent storage to the web server. The local persistence scheme must consider several criteria:

- The persistence scheme must be supported and available in the target platforms the other Endpoint Monitor components function in. As discussed in 3.3.1 the libraries used support x86 and ARM versions of Linux, Windows, and Mac.
- The scheme must be performant to minimize subjects’ overhead and reduce the resources required to run the Endpoint Monitor.
- As discussed in Section 3.4.2, the chunking algorithm used to upload data to the Catalyst web server relies on the assumption that persistence transactions are atomicity, consistency, isolation, durability (ACID) compliant.

³See Java Robot class: <https://docs.oracle.com/javase/7/docs/api/java.awt/Robot.html>.

⁴See JNativeHook: <https://github.com/kwhat/jnativehook>

⁵See OSHI: <https://github.com/oshi/oshi>

The Endpoint Monitor employs MySql or the near-identical MariaDB (depending on what is most easily installed on the install's environment) for local persistent data storage.

3.3.3 Building Installers and Compatibility Issues

Each operating system supported requires a different installer to provision all of the necessary components. Installers may be shared sometimes among similar enough distributions of the same operating system category (Windows, Linux, Debian, Fedora, etc.) but most often each distribution requires its own installer. Installers must:

1. Download system clock synchronization tools (if necessary)
2. Synchronize the system clock
3. Install MySql
4. Download the database configuration
5. Provision MySql with the database structure and users
6. Configure MySql to launch on startup
7. Install Java
8. Download the Endpoint Monitor
9. Configure the Endpoint Monitor to launch on startup with the upload token supplied to the installer on download
10. Restart the device

Currently, installers have been built and tested for Windows 10, Ubuntu Linux (up to 21, though versions beyond 20 require selection of “Ubuntu on Xorg” on sign in to function properly), Kali Linux (up to 2022.1, though currently Kali Linux uses a Wayland-based windowing system incompatible with the latest version of OSHI, so window data collection is not yet reliable.), and Raspbian Linux.

3.3.4 Performance Evaluation

The endpoint collector must perform such that:

1. The device remains usable to subjects.
2. The Endpoint Monitor does not require extensive computing resources subjects may not have.
3. The data maintains sufficiently high fidelity (particularly with regard to screenshot frame rate) to be used for analysis purposes.

Several common software metrics to evaluate performance include (1) CPU time use, (2) memory space use, (3) disk bandwidth use, and (4) network bandwidth use [344]. While other metrics (such as GPU use [360]) may be relevant in specific circumstances, Catalyst in particular operates primarily on the listed device resources. If the Endpoint Monitor functions properly without saturating these performance aspect on common hardware, it likely performs adequately without interfering with user activity.

Catalyst collects running process information at the operating system level; as a result, data collected while running Catalyst can also be used to evaluate its performance. Each process recorded by the Endpoint Monitor contains **CPU** and **mem-**

ory use data.⁶ Users' data, then, depicts how much overhead the Endpoint Monitor and its local storage (MySQL or MariaDB) generate. Modern devices use processors with multiple cores and tend to have large amounts of memory, and professional reverse engineers likely use high-end modern hardware;⁷ using an entire single core, then, would be acceptable. Memory, on the other hand, is measured as a monolithic resource, but using 25% or less of system memory would be acceptable.

Disk bandwidth, in contrast to CPU and memory use, is more difficult to measure. Not only do local storage systems vary drastically in performance [178] but operating systems vary in file system access patterns and methods. Measuring file system access bandwidth in an operating system independent method, then, is problematic; if it were possible to do such a measurement, that measurement would not provide a depiction of system overhead without knowing the capability of the underlying storage hardware and the disk access pattern. [208] As a result, process data collected by Catalyst does not contain useful disk bandwidth data. However, Catalyst does collect some data points which can be somewhat indicative of storage performance: When doing disk writeouts (writing data collected during a time quantum, set at 1 second in Catalyst to storage) the Endpoint Monitor collects a pair of timestamps: The data collection timestamp, generated with a piece of data such as a screenshot or mouse click in the application, and an insertion timestamp, collected in the storage engine when the data is placed in local storage. Monitoring the latencies between the application and storage timestamps can indicate whether the storage system is overtaxed — large and erratic latencies would exist in such a

⁶Process attributes also include the Virtual Memory Size (VSZ) and Resident Set Size (RSS) which refer to memory use but these attributes may not prove accurate for performance analysis, as they refer to allocations and shared memory footprints, which in turn may vary across different environments and with differing user activity.

⁷This assumption was later confirmed based on the architectural data collected during competitions, as seen in Chapter 4.

case. If latencies remain low and constant, then the storage engine keeps up with the data generation and indicates that the system is likely (but not certainly) not lagging based on local persistence.

As with disk bandwidth, **network bandwidth** is difficult to measure in a reliable, environmentally independent way. However, the Endpoint Monitor successfully uploading data provides a positive indication that the system does not overtax network use; unfortunately, if the Endpoint Monitor overuses network resources then it would not be able to upload data and a corresponding negative indication does not exist. Network bandwidth use acceptability, then, relies entirely on user feedback in cases where the Endpoint Monitor attempts to transmit too much data.

3.3.5 Endpoint Monitor Performance Measured in Deployment

After the CTF events summarized in Table 1.1, the collected data demonstrates that the software does not require huge amounts of computing resources and likely does not cause interference in device usability; subjects were instructed to assign at least 3 cores and at least 4GB of memory to their devices but the Endpoint Monitor did not use more than about 110% of a CPU core⁸ and more than 20% of their memory across operating systems (both Linux and Windows based) and architectures (x86 and ARM). Figures 3.4, 3.5, 3.6, 3.7 and 3.8 illustrate these **CPU** and **memory use** patterns (of both Catalyst and its associated MySql/MariaDB processes) from data collected by participants in The Grand Re. These figures graph CPU or memory use over time since startup.

In more detail, the Endpoint Monitor uses around 90-100% of a CPU core con-

⁸On startup, the Endpoint Monitor did use considerably more than 100% of CPU use, but this generally decreased to the typical 100% use. This spike is typical of Java programs, which use significantly greater resources on startup due to launching the Java Virtual Machine and associated initial bytecode interpretation overhead. [126, 259, 189] The temporary overhead is not considered problematic, given that it only effects device usability for a short period of time.

sistently across platforms. MySql, while under load from the Endpoint Monitor, has significant differences in environmental averages but tends to be under 20% CPU use (the highest system average was 17% on Ubuntu) that slowly increases over time due to increasing index size — future work (Chapter 8 discusses this in detail) could remove stale data in order to potentially counter this increase. Some MySql CPU uses are shown on the plot to be significantly larger than average and these should be addressed in future work as well, but they did not appear to have a large impact on the deployment for the reverse engineering events. Memory use depends on the amount of system memory available, since the process monitor expresses this as a percentage of the whole, though as mentioned averages are acceptable on subjects' devices.

Occasional abnormalities in the CPU and memory use data may be due to interruptions such as devices sleeping or changes in incoming data due to factors such as monitor resolution changes. Process data CPU and memory use are recorded as integers, consistent with the data source library and operating systems used, as is apparent in these charts.

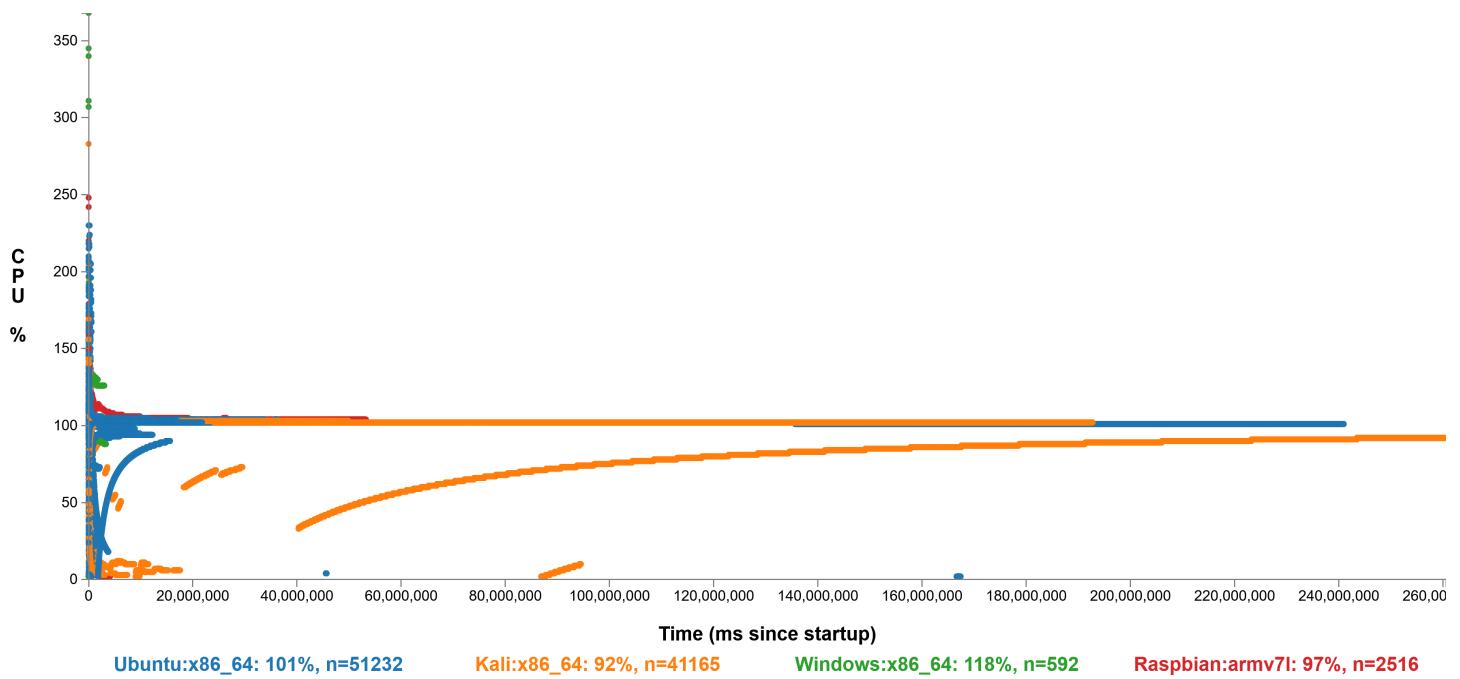


Figure 3.4: The Endpoint Monitor's CPU use over time from participants in The Grand Re Challenge (see 4.7) colored by environment type with averages listed.

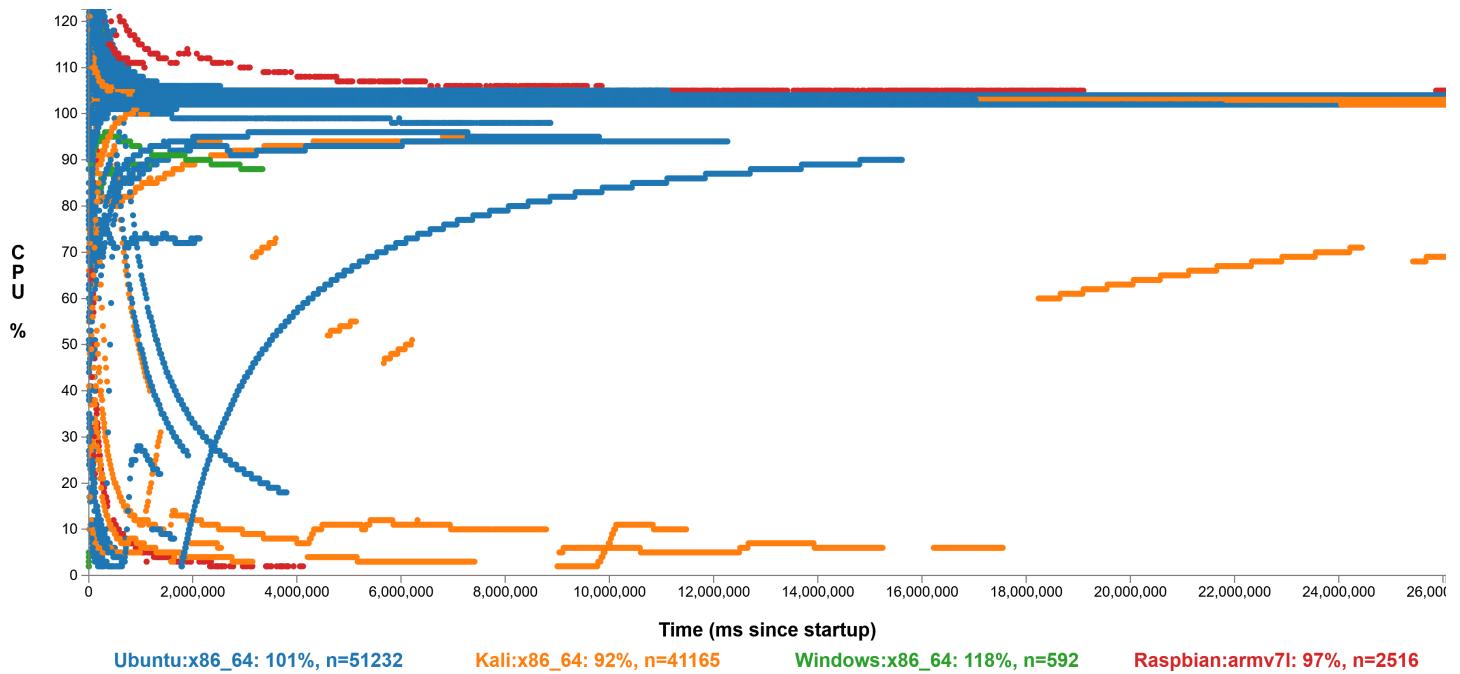


Figure 3.5: A zoomed version of Figure 3.4 which crops most of the startup overhead data points in order to better view patterns in the data.

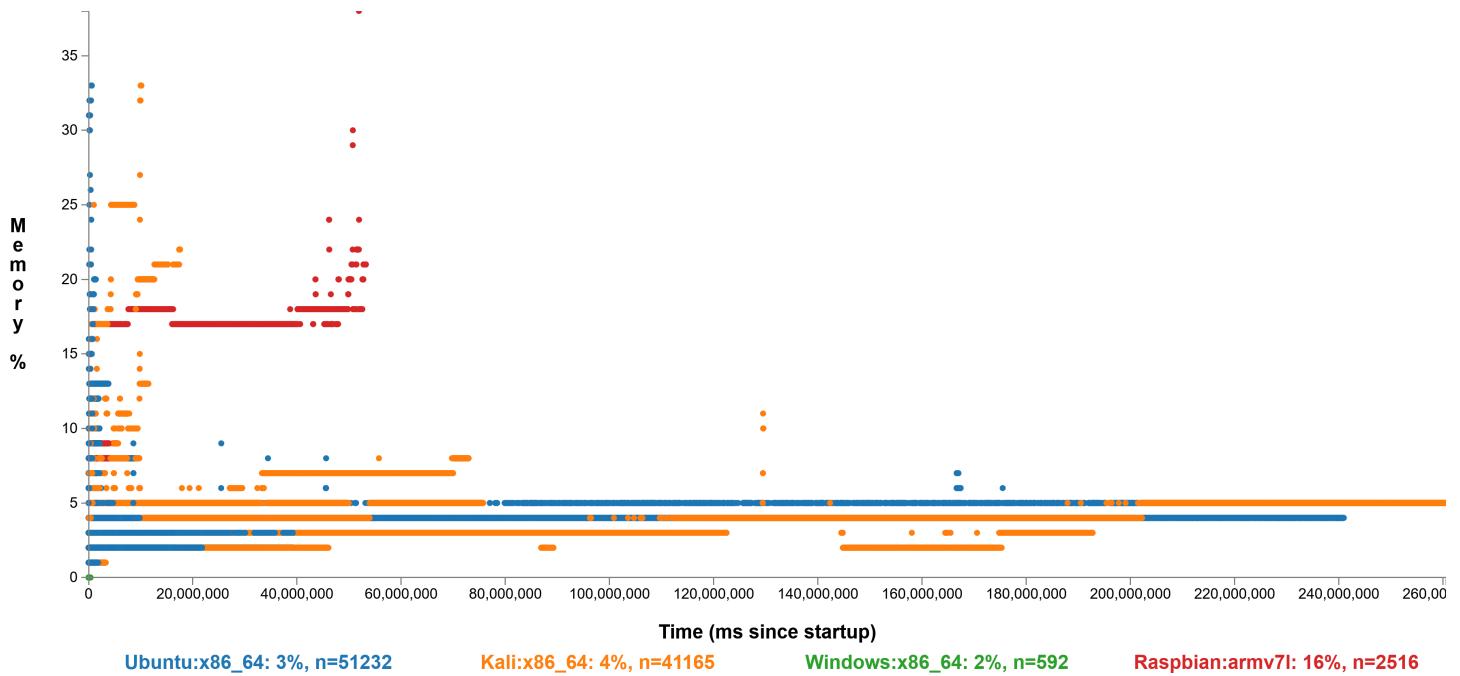


Figure 3.6: The Endpoint Monitor's memory use during the Grand Re.

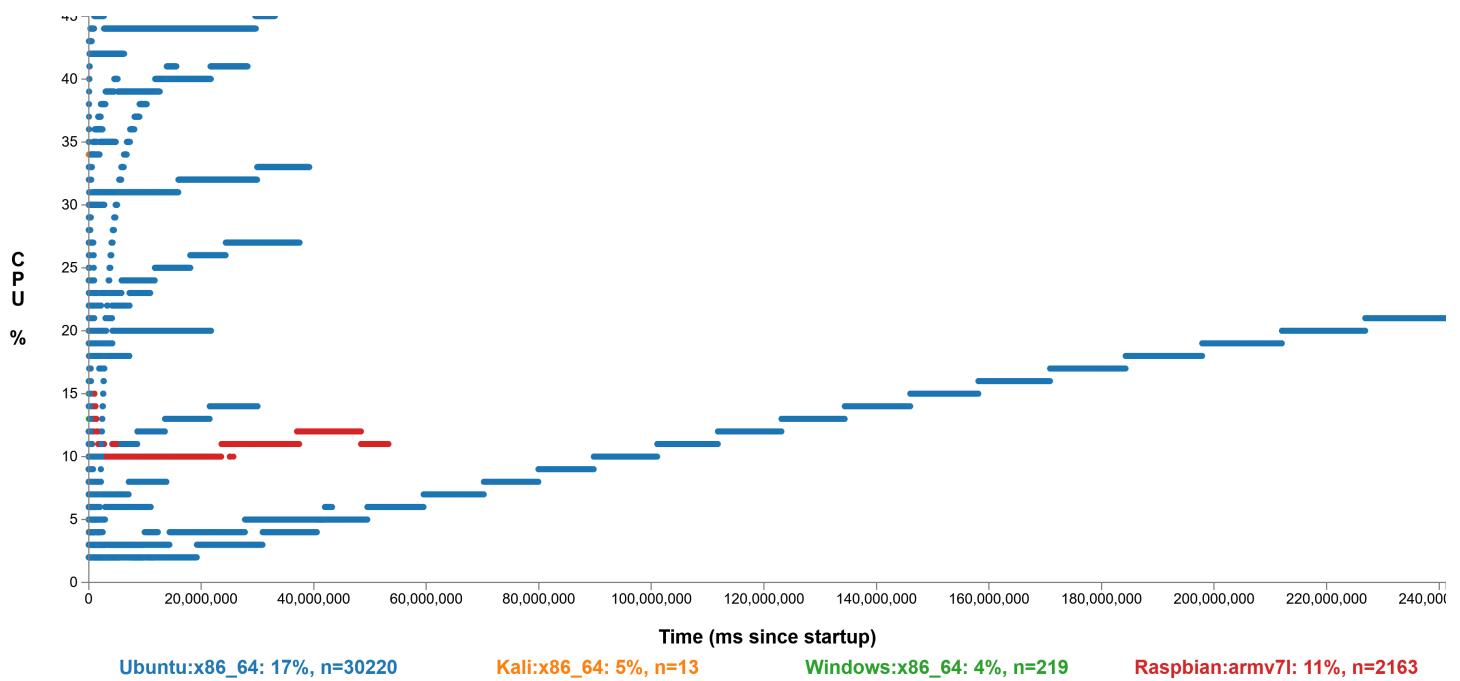


Figure 3.7: The CPU use for MySql, a dependency used by the Catalyst Endpoint Monitor, from participants in The Grand Re Challenge, colored by environment type with averages listed at the bottom.

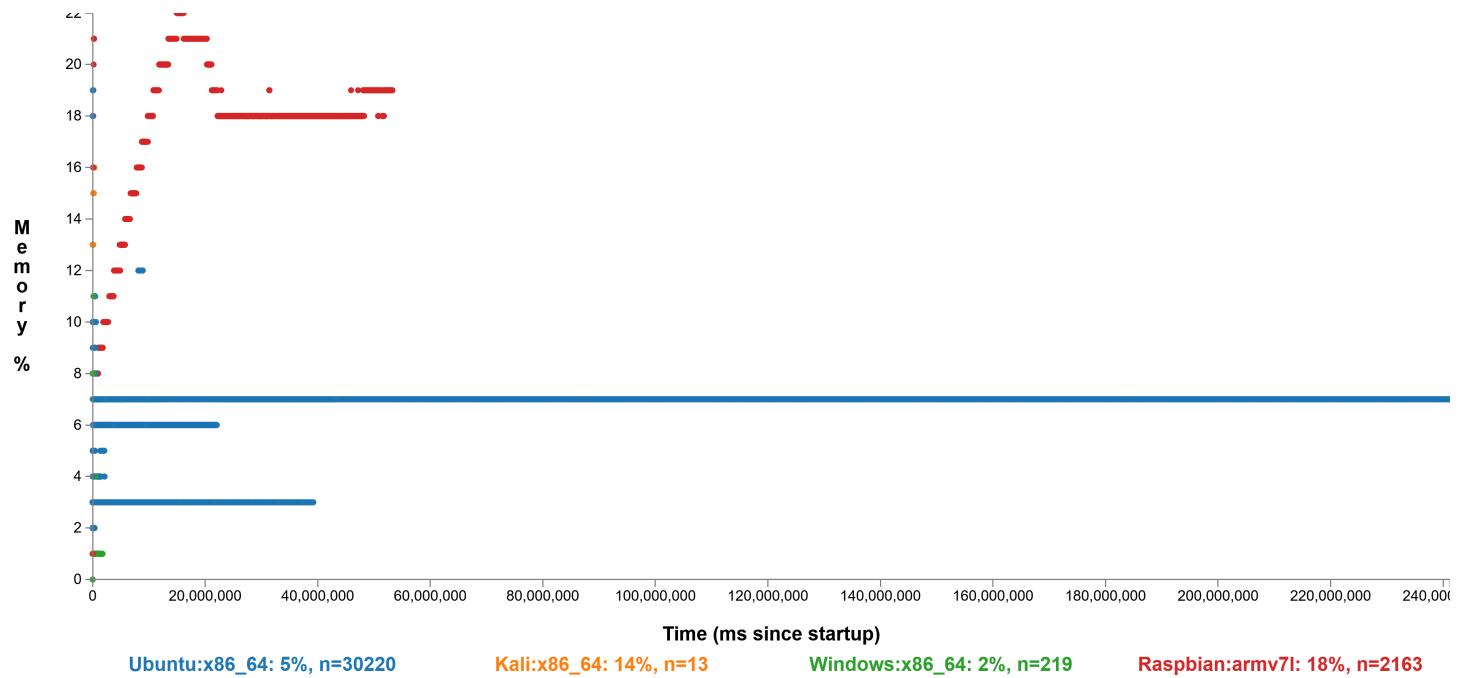


Figure 3.8: MySQL's memory use during The Grand Re.

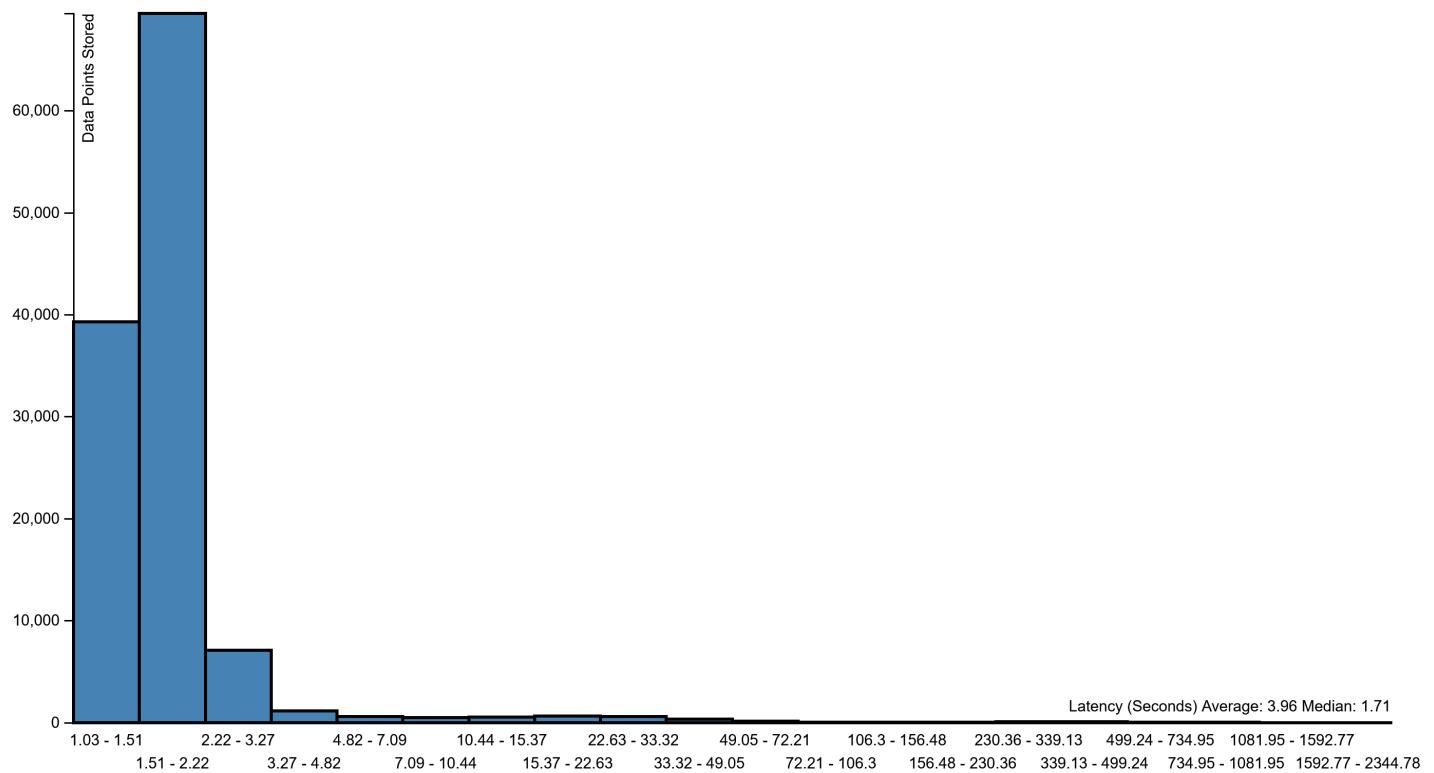


Figure 3.9: This histogram shows the average screenshot insertion and storage latencies recorded on subjects' devices during The Grand Re. The bins are logarithmic scale; while most latencies show less than two seconds, a significant number of operations took more than that.

Additionally, a vast majority of storage operations on users' devices took less than two seconds, as shown in Figure 3.9, which provides sufficient performance for good data fidelity and indicates adequate **disk bandwidth** use. Notably, some sessions encountered very large (orders of magnitude larger than 2 seconds) latencies on a small number of data points. It is not clear what caused these latencies and further investigation may be warranted — they may be due to edge cases such as a device hibernating during the middle of a write, for example. Nonetheless, these datapoints did not appear to have a substantial impact overall — neither users nor analysts provided feedback encountering an issue with device usability or the data recorded.

Subjects did not complain about performance impacts of the Endpoint Monitor, though many subjects downloaded the Endpoint Monitor and did not finish installing it; others downloaded and ran the Endpoint Monitor but did not appear to do much while running it. It is possible that those users found the data collection to be too much overhead and stopped without providing feedback. It is also possible that some subset of users encountered **network bandwidth** issues, thus encountering a silent negative indication, but feedback does not indicate this was the case.

3.4 Data Uploading Method and Performance

After collecting data, the Endpoint Monitor streams that data to the server via a network connection. Already, several algorithms and tools exist to transfer data over the network, generally in the form of files. [313] None of these are well adapted for Catalyst's use. The classic File Transfer Protocol (FTP) [257] sends entire files over standard TCP streams, but can only send the entire file at once (which is inefficient for files that change or update with additional data, as is the case with Catalyst) and has difficulty dealing with network interrupts, particularly with large files. [13]

Additionally, it is difficult to manage an unknown number of users uploading via FTP concurrently on limited server hardware, as discussed in Section 3.4.1.

`rsync` [265] synchronizes files across the network, propagating changes to a file on one device to another by using a rolling checksum algorithm to transfer only file differences rather than the entire file. This algorithm calculates hashes on segments of data over the entire length of the file and compares them to the same hashes on the remote server. Although this algorithm proves efficient in many use cases, for time series, logged data it is not necessary to scan the entire file for changes; the changes all occur at the end of the log. Given data sizes on the scale of gigabytes, then, algorithms like the one employed with `rsync` would cause excessive file scanning and potentially poor performance (dependent on file system access speeds) when synchronizing. Rather than a whole-file-diff system, log files only need their tails sent — the data added since the last successful sync. [290] While Catalyst does not use the filesystem directly (instead using more optimized persistent storage access with in-memory caching and efficient indexing via a local database manager) it employs a similar log sync method.

The tools described also have drawbacks relating to:

1. Operating on the file system
2. Requiring operating system level permissions, and
3. Introducing additional dependencies

Catalyst generates large amounts of data which requires structuring in order to query efficiently; plain log files on the filesystem, even if sorted in order, do not provide sufficient performance for later processing and visualization at scale. Thus,

these files need to be combined and structured in order to use, and simply synchronizing files, rather than synchronizing to structured data, introduces additional steps and complexity as the files will need to be aggregated and structured downstream of the file synchronization.

Using the file system directly also requires that users be given operating system level permissions. While it is possible to do this in a secure way, automatically generating file system permissions for a large number of unknown participants poses significant risk to the entirety of the system. Instead, isolating Catalyst-specific functionality to the Catalyst server (using Catalyst-internal permissions) reduces this risk significantly.

Already, Catalyst requires Java and MySql/MariaDB as dependencies on client devices. In practice, even satisfying these and building installers for different environments proves difficult. Adding more dependencies, as may be necessary for some network transfer tools tools, introduces dependencies which may not be uniform or easy to interact with in all environments. A single, unified solution built into the Endpoint Monitor eliminates this complexity.

3.4.1 Implemented Solution

As shown in Figures 3.3 and 3.2, once an Endpoint Monitor stores HDI data on local persistence, it then streams the data to the back end server. Once on the server, this data resides in a database, ready to be queried and formatted for export, ultimately to be used for visualization and analysis.

A websocket (WS) connection, which employs the transmission control protocol (TCP) at a lower level, controls the low-level transmission details and ensures maximum transmission speed and prevents network data loss. Even with these technologies, the size of the data — over 25 GiB total for 8010 active minutes, or 3.35

MB/s per active user for up to 39 concurrent users in events to date, shown in Tables 4.5 and 4.3 requires the implemented protocol to account for server capability and the potential for participants to encounter lossy, unreliable networks.⁹ On the server side, the potential for a large user base uploading concurrently leads to two different modes:

- Real time uploading as subjects use their devices¹⁰
- Singular uploads activated by a user which send all of the data at once¹¹

The first mode disperses data uploads over a larger time period; rather than having single, large uploads this mode separates uploads into multiple small pieces over a long period of time. In cases where many users might be recording data concurrently, this mode prevents large upload events when users finish their tasks and upload their entire traces at the same time, reducing peak server and network workloads. Additionally, this continuous upload mode reduces the number of steps for participants to upload their data and results more users submitting more data; in cases where participants do not succeed in solving problems and thus do not have as strong incentive to upload their data, they do not often undertake the additional steps to start uploads. This was apparent in the initial RevEngE deployments — subjects who were unable to succeed did not upload their data. The continuous upload mode, by contrast, generated a significant amount of data from unsuccessful

⁹Users might be based in a geographical area [194] where such networks are common; even in more developed areas service outages and degradation still occur from time to time. Additionally, using privacy technologies (as one might expect from professional reverse engineers) such as The Onion Router [250, 14] (TOR) and virtual private networks [188, 225] (VPN) can create network performance issues.

¹⁰This method was used for RevEngE CTF and The Grand Re, where many users were expected to participate concurrently.

¹¹This method was used on the original RevEngE deployments, where participants were expected to finish at different times.

participants later in The Grand Re — data which can be evaluated to determine which reverse engineering strategies did not work well.

Both of these modes utilize the same uploading algorithm; the single upload simply stops uploading when it completes. Even with a WS/TCP stack, both the uploading and exporting components require additional chunking and throttling in order to cope with potentially lossy or slow networks.

3.4.2 Streaming Data to the Server

The Endpoint Monitor accumulates large amounts of data as subjects run it. While the exact amount varies according to the amount of activity and settings¹² the aggregated size of data from the RevEngE and Grand Re Challenge deployments reached around 25 GiB, of which the three successful subjects produced the majority. Individual data, then, scales to multiple GB at least. Given the assumption that network connections may be somewhat slow and lossy, this amount of data cannot easily be transferred in single, monolithic uploads — such uploads can be interrupted by network loss before they complete. Additionally, given this size of data, querying all of it from persistence for an upload can easily be too large for local resources — particularly memory — making it necessary to query and write to the network smaller pieces of data at a time.

In order to resolve these issues, the Endpoint Monitor *chunks* and *throttles* data uploads. It does this based on datapoint timestamps. The algorithm, technically specified in Algorithm 1 and illustrated in Figure 3.10, is as follows:

1. Define a default max interval.

¹²User activity levels that change screenshot, window, and process polling intervals and variables such as number of processes running and display resolution all determine the size of the collected data.

2. Define a change factor which changes the time period chunking depending on conditions (to be described shortly) this algorithm meets during execution.
3. Query the database for the last sent item's timestamp as the minimum query timestamp.
4. Query the database for the current database timestamp as the maximum query timestamp.
5. If the maximum query timestamp (the current timestamp) is more than the minimum plus the max, then assign it the minimum plus max value instead.
6. Query all data points between the minimum and maximum query points and send that to the back end server.
7. If the data uploads successfully, store the maximum query timestamp as the minimum for the next upload. Check memory use by multiplying the current memory allocation by the change factor¹³ and comparing to available¹⁴ memory: If the algorithm does not anticipate memory overflow multiply the max interval by the change factor.
8. If the data does not upload successfully, divide the max interval by the change factor.

¹³Since the change factor is used as a time period selection multiplier, the algorithm assumes that memory use will scale maximally by that change factor: Selecting data from a time period that is twice as large as the previous (assuming a change factor of 2) will require maximally (and ideally) twice as much memory as the previous time period. While this may not hold completely true in all cases if data is distributed highly unevenly in different time periods, in practice this test works and overhead in the initial allocation (which is included in the previous time period allocation metric) provides a bulwark against uneven data distribution.

¹⁴Available memory is the free memory in the Java Virtual Machine (JVM), not total system memory.

Algorithm 1 The client-side, Endpoint Monitor data upload algorithm.

```
 $\Delta T_{interval} \leftarrow \Delta T_{interval_{default}}$ 
 $\Delta T' \leftarrow \Delta T'_{default}$ 
while Running do
     $T_{min} \leftarrow T_{max_{sent}}$ 
     $T_{max} \leftarrow T_{current}$ 
    if  $T_{max} > T_{min} + \Delta T_{interval}$  then
         $T_{max} \leftarrow T_{min} + \Delta T_{interval}$ 
    end if
    Send :  $D_{total}|T_{max} > n_x[Time] > T_{min}$        $\triangleright$  Send all data points in the time given period to the server.
    if Success then                                 $\triangleright$  Success is true on successful reply from the server.
         $T_{max_{sent}} \leftarrow T_{max}$ 
        if  $Mem_{used} * \Delta T_{interval} < Mem_{max}$  then
             $\Delta T_{interval} \leftarrow \Delta T_{interval} * T'$ 
        end if
    else
         $\Delta T_{interval} \leftarrow \Delta T_{interval} / T'$ 
    end if
end while
```

In order for this algorithm to work, all data entries with a timestamp before the current time must be in the database before they are queried for upload; otherwise, the upload algorithm will move onto the next frame, missing data entries with timestamps prior to the new frame minimum, as illustrated in Figure 3.11. This requirement is satisfied by inserting all datapoints with a database-generated timestamp (which has to be earlier than all timestamps produced after, including upload chunk timestamps) and using an atomicity, consistency, isolation, durability (ACID) compliant database. [210] Atomicity ensures that all data entry insertions are done independent of the chunk timestamp generation, while isolation ensures that the timestamps are generated correctly in the order that data points enter the database.¹⁵ Together, these properties ensure that the algorithm works, and the selected database manager (MySql) ensures ACID compliance, where highly performant but non-ACID compliant alternatives such as MongoDB [49] could fail in this role by uploading data time periods before all data points for those periods are readily readable.¹⁶

3.4.2.1 Device Clock Complications

One particularly problematic aspect of the data uploading algorithm is system clock synchronization. MySql uses the system clock to generate timestamps. Thus, the entire algorithm relies on a stable system clock; should the system clock move backward in time unexpectedly, the last timestamp uploaded will likely be forward in time compared to data entering the database. New data points, in this case, will not be uploaded. Different operating systems and different network environments have

¹⁵The other ACID properties — consistency and durability — are useful for maximum data fidelity, but not required for this algorithm nor the Endpoint Monitor generally.

¹⁶It is possible for alternative schemes to work if it is known when consistency is achieved for a given time period — the algorithm would use a "currently consistent" maximum timestamp rather than a "current time" maximum timestamp.

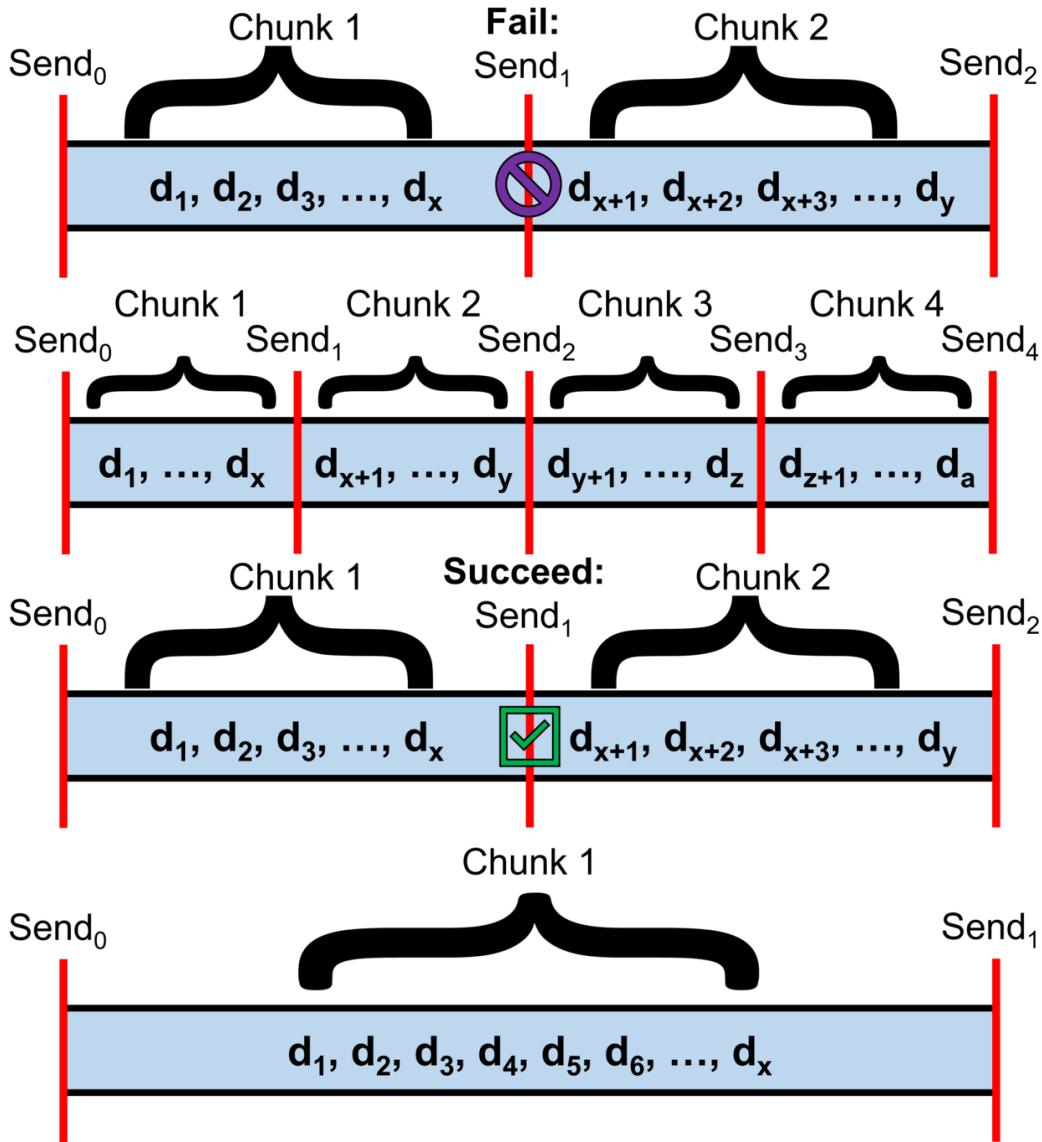


Figure 3.10: An illustration of the upload algorithm showing datapoints arranged on a timeline ($d_1, d_2 \dots d_x$) and attempted uploads depicted by red lines marking upload time periods. When a chunk of data fails to upload, the chunk time interval is halved, resulting in chunk sizes roughly half of the initial size. When the chunk succeeds, the chunking algorithm doubles the time interval.

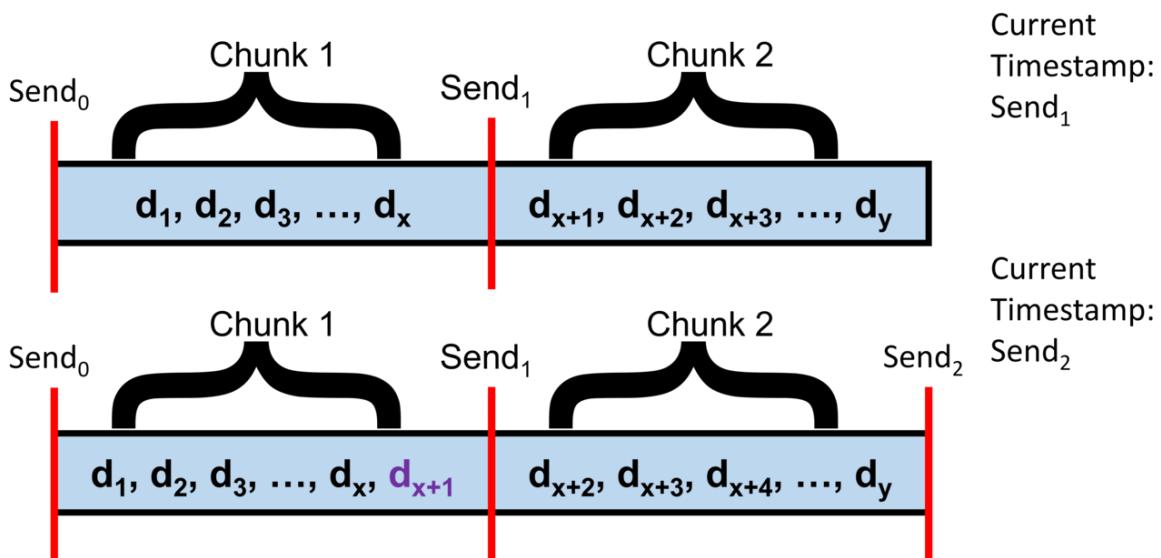


Figure 3.11: The Endpoint Monitor chunks data based on timestamps. Initially, the Endpoint Monitor sends the chunk between $Send_0$ and $Send_1$. Then, it sends the chunk between $Send_1$ and $Send_2$. Datapoints all have insertion timestamps. ACID compliance is assumed here, preventing d_{x+1} from being inserted after a query terminating at $Send_1$, where $Send_1 < \text{Timestamp}_{\text{current}}$

a variety of schemes dealing with system time, many of which receive system clock synchronization. This can (and, in some Catalyst deployments, has) caused issues with uploading. A manual reset of the upload timestamp will restart uploading from the beginning in these cases, but the data collected will even then have timestamp corruption.

3.5 Summary

Catalyst collects user behavior data by monitoring HDI on individual devices and streaming it to a back end server for later visualization and analysis. Doing this presents technical challenges in building a system that (1) works across different platforms, (2) performs adequately enough that it does not significantly interfere with user behavior on common hardware, and (3) streams data to and from a server efficiently and with scalability. Ultimately, Catalyst performed well and collected large amounts of data in several reverse engineering challenge deployments, as discussed in Chapter 4.

Chapter 4

REVERSE ENGINEERING CHALLENGES

Although Catalyst can be used in many applications to extract data from the HDI, I built Catalyst as a tool to monitor users as they completed computer security tasks [308, 309]. Specifically, the research presented in this dissertation aims to gather information regarding reverse engineering and code obfuscation — by using Catalyst to observe individuals reverse engineering binaries — in an attempt to generate an empirical understanding of reverse engineering and code obfuscation techniques. The ultimate goal is to discover how well code obfuscation and tamper proofing protect assets in a program, and, conversely, which reverse engineering methods are most effective in thwarting such protections.

To do this, Christian Collberg and I (the “initial team” or “initial research team”) built three computer security competition engines and hosted reverse engineering problems generated by Tigress — (1) the *RevEngE Pilot*, (2) *RevEngE CTF*, and (3) **The Grand Reverse Engineering Challenge**. The initial *RevEngE Pilot* study served as a pilot for the methodology, demonstrating that (1) the initial team could automatically generate realistic reverse engineering challenges, (2) the Catalyst tooling could be run on users devices — with acceptable overhead — to gather data, and (3) the recruitment methods could attract interest. On the third point, the initial team only achieved modest success. This led to *RevEngE CTF*, and evolution of the *RevEngE Pilot* which fixed technical issues with Catalyst, introduced more user-friendly interfaces and competition modes, enhanced recruitment strategies, and ultimately yielded high quality reverse engineering data. The *Grand Re* picked up

where *RevEngE CTF* stopped, expanding *RevEngE CTF* with more researchers, more tools tested, and more outreach for recruitment; The *Grand Re* succeeded in generating a significant amount of high quality reverse engineering data from two subjects.

In this third and most effective event — the *Grand Re*— the initial research team was joined by additional researchers from Ghent University and Irdeto; the Ghent University team (with help from the initial team) later also deployed Catalyst for a small-scale experiment observing students reverse engineering. Specifically, we were joined by Bjorn de Sutter, Bart Coppens, Bert Abrath, and Jens Van den Broeck from Ghent University; and Catherine Chambers and Phil Eisen from Irdeto in building and deploying obfuscated binaries for participants to reverse engineer (“challenges”) for The *Grand Re*. Later, Waleed Mebane from the University of Arizona and Tab Zhang from Ghent University joined during the analysis phase of the event.

The *Grand Re* included problems generated with other tools developed by the research teams from the ASPIRE project and Irdeto. Monetary prizes were awarded those who succeeded and provided data with Catalyst. Prize details for each event are shown in Table 4.6. This expanded on Christian Collberg’s prior efforts to gather reverse engineering data; these earlier efforts did not include Catalyst but did generate results. Though lacking the low level HDI data, the one of the subjects published their methods and subsequent work breaking the challenges (Salwan [278, 277]).

This chapter first acknowledges IRB approval and funding details in Section 4.1, then discusses types of challenges in Section 4.2 before describing subject recruitment methods in Section 4.4. The chapter then focuses on details of individual events, detailing the *RevEngE Pilot* study in Section 4.5, *RevEngE CTF* in Section 4.6 and The *Grand Re* in Section 4.7. Finally, the chapter summarizes the results of these events in Section 4.8 and discusses a subsequent Ghent University study in Section

4.9.

4.1 Study Acknowledgements

The studies described here were approved by the University of Arizona Institutional Review Board as human subject research. Details can be found in Appendices B and C. Additionally, the event hosting and data analysis efforts were supported in part by NSF grants 1145913 and 2040206. Additional non-grant, unrestricted funding was used for some of the prize money. The availability of such funding sources can be important as funding restrictions may not allow subjects to be paid large sums or university regulations may make it problematic to pay non-US citizens.¹

4.2 Challenge Types

The research here aims to gauge practical software security protection methods and evaluate reverse engineering methods. To do this, challenges must — as much as possible — mirror real-world software protection and reverse engineering tasks. Based on previous experience from Tigress [94, 278] and ASPIRE-based [80] challenges, we identified three types of reverse engineering tasks representative of real-world practices:

1. **Extraction** where reverse engineers extract a particular piece of data from a program.
2. **Tampering** where reverse engineers change the execution of a program, generally in order to bypass password/key/security checks.

¹We experienced such issues when the University of Arizona demanded that subjects provide their banking details and subjects refused this due to privacy reasons.

3. **Deobfuscation** where reverse engineers remove obfuscations from a program, attempting to revert it to a nonobfuscated form.

These follow real-world use cases: Reverse engineers obtain license keys or media assets from programs, as is the case for extraction challenges; they also attempt to bypass activation-based protections such as license key checks, as is the case in tampering. Finally, they sometimes try to extract adversaries' code in order to use it for their own purposes, a problem presented by deobfuscation challenges.

Even with the notion of real-world use cases, these challenges do not necessarily parallel real-world program protection or reverse engineering practices perfectly. Most obviously, real world software tends to be much larger than the challenges generated here, although their security sensitive kernels can be relatively small. Future work should involve conducting formal comparisons between these challenges and real-world practices with methods such as surveying reverse engineers who have real-world and synthetic challenge solving experience.

4.3 Submission Evaluation

Once participants complete challenges the system must determine if what is submitted is, in fact, a solution to the challenge. The participants submit their solutions either via the competition engine — in the case of RevEngE variants — or via email — in the case of The *Grand Re*. The former conducts automated submission analysis while in the latter case the researchers manually checked solutions. In both cases, the three types of challenges require different types of evaluation.

Evaluating extraction challenges for correctness is trivial: if the submission is identical to the asset the research team embedded in the challenge program, then the submission is correct.

Evaluating the correctness of tampering and deobfuscation challenges is more difficult: In both these cases, functional equivalence (or at least an approximation thereof, as program equivalence is a known hard problem [158]) must first be established between submissions and original/expected program behavior, and in the latter case some additional criteria must discern whether the submitted code is “close enough” to the original to be deemed a successful deobfuscation [288, 309].

4.3.1 Functional Equivalence

This work approximates functional equivalence by using test cases, both randomly generated and automatically generated with symbolic execution using klee [63] as detailed in Algorithm 2; steps listed here refer to the numbers in that algorithm. (1) This symbolic execution finds all paths through the original program and generates a list of test cases which result in high code coverage — these cases, ideally and if klee functions perfectly, result in the same output expression for all values between sorted pairs of test cases. After klee generates test cases for a program based on original, non-obfuscated code, (2) an evaluation program determines whether the original and the submission function the same at test cases and (3) at the average in between all consecutively ordered test cases — average cases are generated by pairing each edge case value with the next value, where all values are sorted, and taking the average between each pair; comparing the output expressions for equivalence is a known hard problem so using concrete inputs is done as an approximation [212].

This does not ensure complete functional equivalence, however: If a submitted program has different test cases than the original, for instance, it is not functionally equivalent. Symbolic execution to determine test cases requires a significant amount of analysis and is not always feasible since heavily obfuscated code has the potential to cause path explosion [347]. (4) Randomized testing with cases on different orders

of magnitude (1 random case in each range of -10^x to $+10^x$ for all x such that $10^x < \text{input}_{\max}$) is also done to help catch non-equivalent cases.²

Tampering challenge submissions that are functionally equivalent are considered correct. Deobfuscation challenge submissions that are functionally equivalent are then analyzed to determine whether or not the submission has successfully removed obfuscation(s) as well.

²Random values average to around half of the max, so testing only random values generated on the magnitude of the max value is likely to miss control flow based on smaller input values; intuitively, programs often do not expect values near the maximum as input.

Algorithm 2 The algorithm employed to approximate functional equivalence. $P_{original}$ is the non-obfuscated program a challenge is based on. $P_{obfuscated}$ is the challenge binary distributed to participants. $P_{submission}$ is the program that participants submit to solve a challenge. This code returns *Fail* if the submission is not equivalent to the original program and *Success* if the submission is approximately equivalent to the original. The *Random* function returns a random value between the first and second arguments.

```

//(1) Get test cases for the original program.
 $S_{klee} \leftarrow KleeEdge(P_{original})$ 
 $S_{klee} \leftarrow Sort(S_{klee})$ 
//(2) Test for equivalence on all test cases.
for  $Test \in S_{klee}$  do
    if  $P_{original}(Test) \neq P_{submission}(Test)$  then
        return Fail
    end if
    if EXISTS  $Test_{prev}$  then
        // (3) Get values in between test cases; test cases are assumed to be sorted so that supplemental
        cases include all averages between klee-generated edge case ordered pairs.
         $S_{supplemental} \leftarrow S_{supplemental} \cup Test + Test_{prev}/2$ 
    end if
end for

```

```
//(4) Generate random inputs on different orders of magnitude.  
//The max input is determined by type limits or a known max value for a challenge.  
Limit  $\leftarrow Input_{max}$   
Iterator  $\leftarrow 1$   
while Iterator  $\leq Input_{max}$  do  
    //For unsigned inputs, 0 is used instead of  $-Iterator$ .  
     $S_{supplemental} \cup Random(-Iterator, Iterator)$   
    //This loop gets random values for each power of 10 order of magnitude. Other orders of magnitude  
(such as powers of 2) would also work.  
    Iterator  $\leftarrow Iterator * 10$   
end while  
//Evaluate equivalence on all supplemental test cases.  
for Test  $\in S_{supplemental}$  do  
    if  $P_{original}(Test) \neq P_{submission}(Test)$  then  
        return Fail  
    end if  
end for  
return Success
```

4.3.2 Code Equivalence

In order to determine whether a submission has successfully deobfuscated code, analysis methods can employ static and/or dynamic analysis methods coupled with either manual or (as shown in other work) automated classification. This classification aims to determine if a program is “human understandable” as would be expected for nonobfuscated code [211]. The work here uses manual comparisons based on statically generated CFGs, qualitative human code analysis and comprehension, and dynamically generated instruction counts [307].

Algorithm 3 details the automated instruction count comparison procedure, which is run in real time on RevEngE as subjects upload submissions following the functional equivalence evaluation. (1) The cases generated in the functional equivalence algorithm are stored and reused to count instructions with good code coverage. (2) Then the instruction counting tooling is provisioned to the original, obfuscated, and submitted code. (3) The server runs each test case on each of the provisioned programs, and (4) compares the categorized (arithmetic, control flow, memory, or structure) instruction counts. Because this involves running unknown executables, RevEngE runs programs in a sandbox (specifically Firejail) [202].

This tooling generates dynamic instruction counts using a LLVM [172] compilation pass, which inserts instruction counting code in the LLVM intermediate representation (IR) before final compilation to binary. The inserted code counts IR instructions, as opposed to underlying architecture instructions. LLVM IR is a concise reduced instruction set computer (RISC) architecture with 52 different instructions. Because LLVM IR instructions do not contain the breadth of large complex instruction set computer (CISC) architectures such as x86 — which has on the order of 1000s of instructions — the LLVM IR instructions can more easily be classified by

type such as arithmetic operations, memory access, program structure, or control flow.³

Comparing instruction categories rather than individual instructions is important, as similar or identical functionality can be achieved using variants of instruction types — such as using a switch instead of a branch instruction. Reverse engineers may select one of many different equivalent instructions when deobfuscating a program but deobfuscated programs ought to have similar levels of arithmetic, control flow, and memory access compared to the original and different levels compared to obfuscated code.

Thus, categorical instruction comparison (rather than all-instruction-total or per-instruction comparison) should more accurately gauge high-level similarity between non-obfuscated, obfuscated, and submitted code. Code which performs closer to the obfuscated than the original is considered still obfuscated. In cases where the obfuscated version cannot run the instruction counter,⁴ alternative algorithm metrics compare only the original to the submission.

³The LLVM IR instruction page (see <https://releases.llvm.org/2.6/docs/LangRef.html>) contains classifications for operations, but the algorithm presented here further aggregates some of these classifications into even fewer categories.

⁴In some cases, the byproducts of obfuscation such as indirect jumps (which make CFG generation difficult) causes the LLVM optimizer to fail and the code cannot be instrumented with an LLVM optimization pass as done here.

Algorithm 3 The comparison algorithm which determines if a submission is likely deobfuscated. $P_{original}$ is the non-obfuscated program a challenge is based on. $P_{obfuscated}$ is the challenge binary distributed to participants. $P_{submission}$ is the program that participants submit to solve a challenge. B denotes a compiled version of P provisioned with a LLVM-based instruction counter. O denotes the program counter output of a program run.

```

//(1) Get all cases from Algorithm 2.
 $S_{all} \leftarrow S_{klee} \cup S_{supplemental}$ 
//(2) Generate instruction-counter tooled versions of the original, obfuscated, and submitted programs.
 $B_{original} \leftarrow \text{LLVM}_{Counter}(P_{original})$ 
 $B_{obfuscated} \leftarrow \text{LLVM}_{Counter}(P_{obfuscated})$ 
 $B_{submission} \leftarrow \text{LLVM}_{Counter}(P_{submission})$ 
//(3) For each testing case, get the instruction count from the original, obfuscated, and submitted programs.
for  $Test \in S_{all}$  do
    //Original and obfuscated instruction counts can be cached in practice.
     $O_{original} \leftarrow B_{original}(Test)$ 
     $O_{obfuscated} \leftarrow B_{obfuscated}(Test)$ 
     $O_{submission} \leftarrow B_{submission}(Test)$ 
    // (4) For each test output, determine if the submitted program's instruction count profile (counts categorized according to instruction  $Type$ , which can be Arithmetic, Control Flow, Memory, or Structure) is closer to the original or obfuscated program. The test fails in the latter case.
    for  $Type \in InstructionTypes$  do
        if  $|O_{obfuscated}[Type] - O_{submission}[Type]| \leq |O_{submission}[Type] - O_{original}[Type]|$  then
            return Fail
        end if
    end for
end for
return Success

```

Subjects ultimately only submitted one deobfuscation challenge on which the team could test these methods, but they proved effective on it on that single data point. Table 4.1 shows the instruction counts and comparison between the original and submitted code for this challenge submission; the submitted code outperformed the original code by reducing total instruction count, as well as reducing counts in all categories except for arithmetic instructions. For this case, the obfuscated code could not be performance tested due to the issues with LLVM optimization passes as discussed, but the obfuscated code likely performs worse than the original and submitted code — though the exact amount is unknown. Future work should examine tooling which can handle obfuscated code (which is challenging for the same reasons disassembling that code can be difficult). For example, the code could be instrumented at the binary code level rather than at the intermediate code level (using a tool such as Pin[219], for example), and the resulting machine code instruction traces could be classified and summarized.

Case	Total	add	and	bitcast	fadd	fmul	fptosi	fptoui	fsub	icmp	lshr	sext	trunc	urem	zext
Original	81	3	1	16	22	13	1	1	12	6	1	4	0	1	0
Submission	262	0	0	202	28	14	0	0	17	1	0	0	0	0	0
Diff	181	-3	-1	186	6	1	-1	-1	5	-5	-1	-4	0	-1	0
Percent	223.5	-100	-100	1162.5	27.3	7.7	-100	-100	41.7	-83.3	-100	-100	0	-100	0
Type	Ari	Ari	Ari	Ari	Ari	Ari	Ari	Ari	Ari	Ari	Ari	Ari	Ari	Ari	Ari

Case	Total	br	switch	Total	alloca	getptr	load	store	Total	call	ret	unrea	Total
Original	13	12	1	630	56	187	254	133	30	24	6	0	754
Submission	1	1	0	277	66	1	123	87	5	4	1	0	545
Diff	-12	-11	-1	-353	10	-186	-131	-46	-25	-20	-5	0	-209
Percent	-92.3	-91.7	-100	-56	17.9	-99.5	-51.6	-34.6	-83.3	-83.3	-83.3	0	-27.7
Type	Flo	Flo	Flo	Mem	Mem	Mem	Mem	Mem	Str	Str	Str	Str	All

Table 4.1: Instruction counts and comparison for the deobfuscation challenge broken into two tables. Instruction categories are shown by color (Arithmetic is blue, Control Flow is orange, Memory is green, and Structure is yellow) and differences between the submission and original are shown as an absolute instruction count (Diff) and as a percentage of the original (Percent). Green or yellow cell coloring in these rows indicates an improvement over the original (with fewer instructions executed) and red indicates a performance decrease.

4.4 Announcement and Advertisement Methods

Generating results from these events hinges on attracting qualified reverse engineering experts capable of solving challenges. In order to entice participants, the research team and I used the same set of advertisement strategies for all events coupled with varying prize and compensation amounts:

- Social media announcements on personal and institutional accounts, using the following platforms:
 - Twitter
 - Instagram
 - Facebook (including relevant groups)
 - Reddit
- Brief pieces submitted to Hacker News [32].
- Mass messages sent via reverse engineering and computer security email lists (including an email list of known security professionals)
- Informing students in computer security courses about the events

4.5 *RevEngE Pilot Study*

The first effort to collect reverse engineering effort consisted of The Reverse Engineering Engine (RevEngE) [309, 307].⁵ I initially built RevEngE to monitor students as they completed reverse engineering assignments for a computer security course, and later generalized the engine to offer reverse engineering challenges to the public at large.

⁵See <http://revenge.cs.arizona.edu/RevEngE/index.jsp> and <http://revenge.cs.arizona.edu/RevEngECompetition/index.jsp>

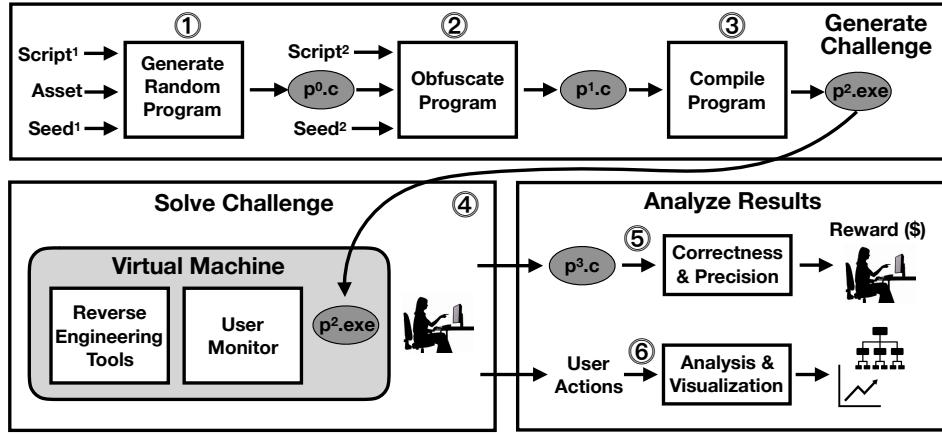


Figure 4.1: The RevEngE system generated challenges automatically from Tigress configuration scripts, served the resulting obfuscated code, and graded results that subjects submitted. Diagram reproduced from original in [307].

The RevEngE system uses Tigress, a C code obfuscator with random program generation capabilities, to generate and obfuscate code. The generation capability builds a program with random hash functions; arguments passed to the generator determine the size and complexity of this program.⁶ These synthetic programs may be protected via different types of checks such as a command line password or a date/time expiration check. RevEngE (in its public facing, competition and CTF modes) used three different sets of arguments to generate the initial challenge programs; cracking type challenges received variants of these with password checks for subjects to find or disable while deobfuscation challenges received versions with no checks since the goal of that challenge type only involves removing obfuscation.

Two of the generation scripts used in RevEngE challenges that a subject solved

⁶Details regarding this generation capability may be found on the Random Function transformation page of the Tigress website: <https://tigress.wtf/randomFuns.html>

can be seen in Appendices G, H, and I. The program generation script is the first tigress call in these scripts. Note that Appendices G and H have the same program generation script and are versions with a password check — the "RandomFunsActivationCodeCheckCount=1" argument creates the password check. Appendix I contains a version for a deobfuscation challenge.

After initial generation, the random hash function programs are then obfuscated, also using tigress, which takes a series of arguments which determine the transform(s) tigress applies to the target source. Typical tigress arguments are listed in Appendix F. Figure 2.2 illustrates an example tigress script which encoded arithmetic, virtualization, and self modifying transforms. Applying this script to the C code in Figure 2.1 yields the obfuscated code in Figure 2.3. Appendices G, H, and I contain tigress scripts used to generate RevEngE challenges that a subject solved.

After the obfuscation step, the obfuscated source may be either released as source code to deobfuscate or compiled binaries to crack or decompile and deobfuscate. Compiled binaries may then be stripped of symbols using the gcc *strip* utility [245].

Subjects then download and reverse engineer the code while running Catalyst and then submit the resulting deobfuscated/decompiled code to the system. Then the RevEngE system determines whether programs are (at least approximately) equivalent in function using Algorithm 3, and, in case of deobfuscation challenges, if they are close in complexity to the original program, using Algorithm 2. Finally, the initial research team reviewed submissions that pass these tests to determine whether they solved the challenge.

The RevEngE system was the first attempt to deploy Catalyst on a reverse engineering challenge with prize money. Prizes theoretically ranged from \$50 USD to \$1000 USD, with prizes increasing with challenge difficulty. However, part of the purpose for this research is to determine how difficult different obfuscation techniques

are to overcome; the lack of compelling data on the subject made it difficult to assign difficulty levels to problems.

Ultimately, RevEngE hosted 50 problems for the public at large, and students from 3 different computer security courses were given the option to participate by running Catalyst while doing assignments.

4.5.1 *RevEngE Pilot* Results

The initial *RevEngE Pilot* efforts did not prove fruitful in generating reverse engineering data; while 69 users signed up for the challenges on the web application, only 4 of those users downloaded the data collection software. None submitted any successful solutions, and none submitted any data collected. Several issues could have prevented participation, including bugs or performance issues in the data collection software,⁷ a data collection software upload mode that users may not have understood, too little prize money, too little publicity, non-compelling competition rules, and proprietary method confidentiality, among other possibilities. In social media announcements, one comment stated that the posting user only worked in Windows environments, so operating system support in Catalyst appears to have had an impact.

The *RevEngE Pilot* did demonstrate that unique and randomized reverse engineering problems — complete with semi-automated evaluation methods — could be generated quickly and automatically on the RevEngE webapp. Additionally, work on Catalyst at this point demonstrated that the target data sources could be collected with high fidelity and acceptable overhead in testing environments with modest hardware in common operating systems. All of these lessons eventually led to *RevEngE*

⁷During this time period, major Wayland adoption and updates to dependency packages in Linux environments could have created unforeseen issues in Catalyst.

CTF, the next phase in the RevEngE event series.

4.6 *RevEngE CTF*

Undeterred by the lack of participation in the *RevEngE Pilot* study, the initial research team made changes to the strategy, addressing potential issues with the initial RevEngE deployment. The changes were based on the data collected related to participant sign ups, aiming to encourage participation among potential subjects who went far enough to download the Catalyst installer for the *RevEngE Pilot*. These efforts resulted in *RevEngE CTF*, built on top of the original RevEngE platform.

Instead of hosting a large number of problems with prizes for solving individual challenges, *RevEngE CTF* consisted of several rounds with a smaller number of challenges, large prizes for top scorers rather than individual challenges, and a time limit for each round; round details are shown in Table 4.2. The initial team supposed that this mode would be less overwhelming and provide a more guided experience, consistent with ideas from prior work hosting CTF events [193, 93]. Additionally, the user interface received a streamlining update, and Catalyst was reconfigured for real-time data uploading in order to maximize user friendliness.

Round	Start	End	Duration	Prize	Challenge Count
1	4/23/2020	5/7/2020	14	\$1,000	30
2	5/8/2020	5/22/2020	14	\$2,000	31
3	5/29/2020	6/5/2020	7	\$2,000	18
4	6/12/2020	6/19/2020	7	\$2,000	3

Table 4.2: *RevEngE CTF* events, durations, and prizes.

The improvements did not immediately yield results: Though several subjects managed to generate traces, no participants submitted a single solution in 5 rounds

of *RevEngE CTF* with \$1000 in prizes for each round. Pivoting further, we reached out to those who had downloaded Catalyst installers by that point, offering each of 12 subjects \$500 to solve individual problems and \$250 for attempts which demonstrated a legitimate attempt without a successful submission.

4.6.1 *RevEngE CTF* Results

All of this proved somewhat fruitful. As shown in Table 4.3, 21 potential subjects signed up for *RevEngE CTF* and 5 users ran the data collection in 21 sessions to generate almost 1500 active minutes of high fidelity data traces. Unlike the initial RevEngE event, one subject solved 3 problems. The exact scripts used to generate those problems is available in Appendices G, H, and I. The original, obfuscated, and submission control flow graphs for one challenge (the deobfuscation and decompilation challenge generated by the script in Appendix I) are shown in Appendix J.

Event	Users	Devices	Total Tokens	Sessions	Total Time	Active Time
RevEngE Pilot + CTF	5	46	21	21	136349	1480
Grand Re Challenge	39	77	96	112	83504	6530

Table 4.3: The statistics regarding user participation for each event: The count of users who participated, the count of the times they downloaded the data collection software, the total number of tokens issued for the event, and the total and active time recorded, in minutes. The number of times downloaded indicates the number of devices subjects attempted to install the data collection on; the number of tokens indicates how many subjects signed up to participate in each event. Active time is defined as the sum of 5-minute intervals where there exists at least 1 user input from the keyboard or mouse.

These results furthered the positive findings of the *RevEngE Pilot*— subjects could be recruited to solve realistic reverse engineering challenges while generating

data with Catalyst. In order to garner more visibility, the initial team reached out to additional researchers in order to host a larger event with more tools to generate challenges and more recruitment effort. The effort produced *Grand Re*, which we will discuss next.

4.7 The *Grand Re* Challenge

Following the initial but limited success of *RevEngE CTF*, the initial research team collaborated with researchers from Ghent University and Cloakware/Irdeto to expand on these previous efforts with a new event, The *Grand Re*.⁸ Aiming to significantly increase subject counts and challenge diversity, we offered a large reward (see Table 4.6) for the competition winners and introduced additional obfuscation techniques to the challenges.

The competition included two rounds, with 7 challenges in the initial round, shown in Figure 4.3, and 3 challenges in the second, shown in Figure 4.4, all of which are summarized in Table 4.4; challenge types included extraction, deobfuscation, and tampering on both x86 and ARM platforms. Extraction challenges involved finding data within the obfuscated programs, while deobfuscation challenges required removing obfuscating transforms and generating non-obfuscated versions of the supplied binaries. Finally, tampering challenges required subjects to make changes to the obfuscated program and submit the altered binary. Obfuscation levels (which we expect to determine difficulty levels) varied from very light to heavy.

Tigress-obfuscated problems used the same generation process as the RevEngE challenges — generating a random hash function with or without a protection and obfuscating that random program with a set of Tigress transforms. The ASPIRE-generated problems, on the other hand, applied ASPIRE protections and obfusca-

⁸See <https://grand-re-challenge.org/index.html>.

tions to existing programs in order to generate challenges. Irdeto generated challenges by building custom programs specifically for *The Grand Re* and applying their in-house tooling to protect them.

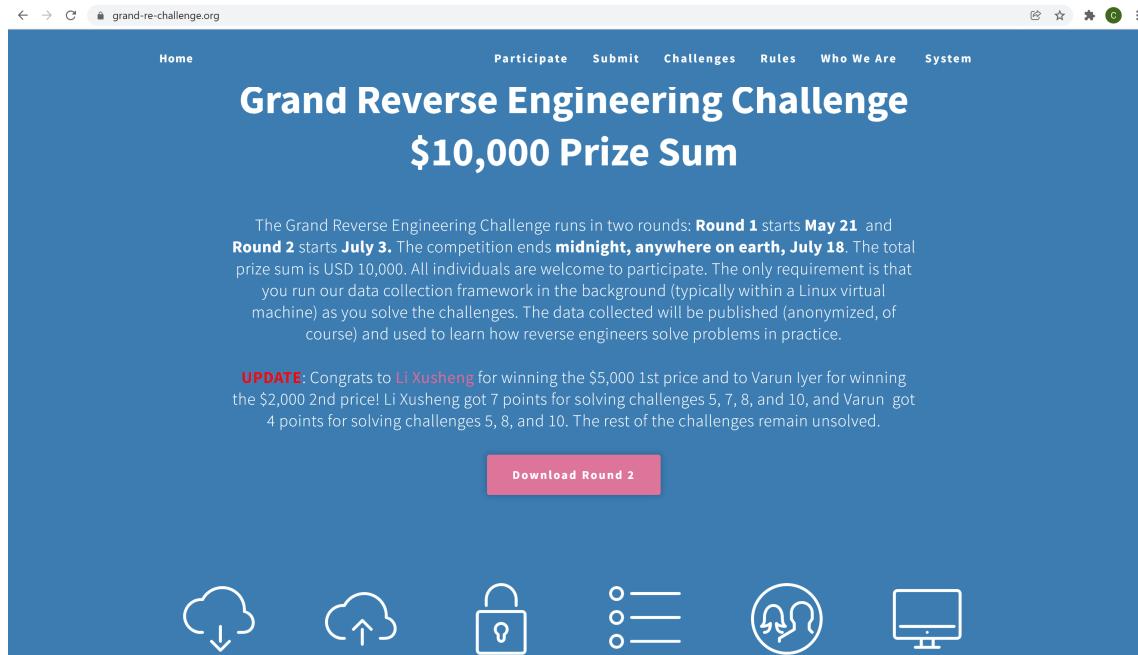


Figure 4.2: The home page for *The Grand Re* Engineering Challenge.

4.7.1 The *Grand Re* Results

The research team produced *The Grand Re* with the lessons from the RevEngE events in mind, leveraging a larger team to increase recruitment efforts and generate a larger variety of challenges while keeping the overall challenge count low as to avoid overwhelming participants. These efforts generated positive results.

The Grand Re attracted more attention than prior RevEngE-based efforts, with 39 users signing up and downloading Catalyst onto 77 devices, as shown in Table 4.3. The subjects uploaded data throughout the event time period, with little pattern to

Round 1

- **Challenge 1** (LIGHT EXTRACT X86 ARM): challenge-1-x86_64-Linux.exe and challenge-1-armv7-Linux.exe read two unsigned 32-bit numbers in decimal form from standard input and print a number to standard output. A successful submission consists of two 32-bit numbers that, when read from standard input, cause the program to crash with a segmentation fault. The original program is about 600 lines of C, the obfuscated program about 21,000 lines.
- **Challenge 2** (MEDIUM DEOBFUSCATE X86): challenge-2-x86_64-Linux.exe and challenge-2-armv7-Linux.exe read two unsigned 32-bit numbers in decimal form from the command line and print a number to standard output. The goal is to deobfuscate the program and return it to idiomatic C. The original function is small, less than 100 lines of C.
- **Challenge 3** (HEAVY DEOBFUSCATE X86): challenge-3-x86_64-Linux.exe takes 3 command line arguments, 32-bit unsigned integers in decimal form. For example, "challenge_3_linux_x86_64.exe 35329 234792379 1100292587" will produce an output of the form "Answer for x = 35329, y = 234792379, z = 1100292587 is 3546740429". The code defines a piecewise mathematical function on the inputs. Your job is to figure out what that function is. The answer is not elegant, but it is short: each part of the function can be written in one line.
- **Challenge 4** (HEAVY DEOBFUSCATE X86): Challenge 4 (challenge-4-x86_64-Linux.exe) is identical to Challenge 3, but uses a different type of protection.
- **Challenge 5** (LIGHT EXTRACT ARM): challenge-5-armv7-Linux.exe takes a license key as its first argument on the command line. It is your job to find this key.
- **Challenge 6** (MEDIUM EXTRACT ARM): challenge-6-armv7-Linux.exe is a protected version of the zip utility. A list of command-line arguments can be generated with "./challenge-6-armv7-Linux.exe --help". An additional, undocumented command-line argument (-K "<some string>") can be used to pass a key to the program. The program only produces valid zip-files when this key is correct. It is your job to find the correct key.
- **Challenge 7** (LIGHT TAMPER ARM X86): challenge-7-x86_64-Linux.exe and challenge-7-armv7-Linux.exe read two English words from standard input and print a number to standard output. The words are no longer than 12 characters, consist of the lower case letters a-z, and have been taken from this [list](#). For two particular words the program will crash with a segmentation fault. Your task is to modify the program to remove the code that causes this crash (it will now print a number to standard output) while maintaining the same behavior as the original program for all other inputs.

Figure 4.3: Challenge descriptions for The *Grand Re* round 1.

Round 2

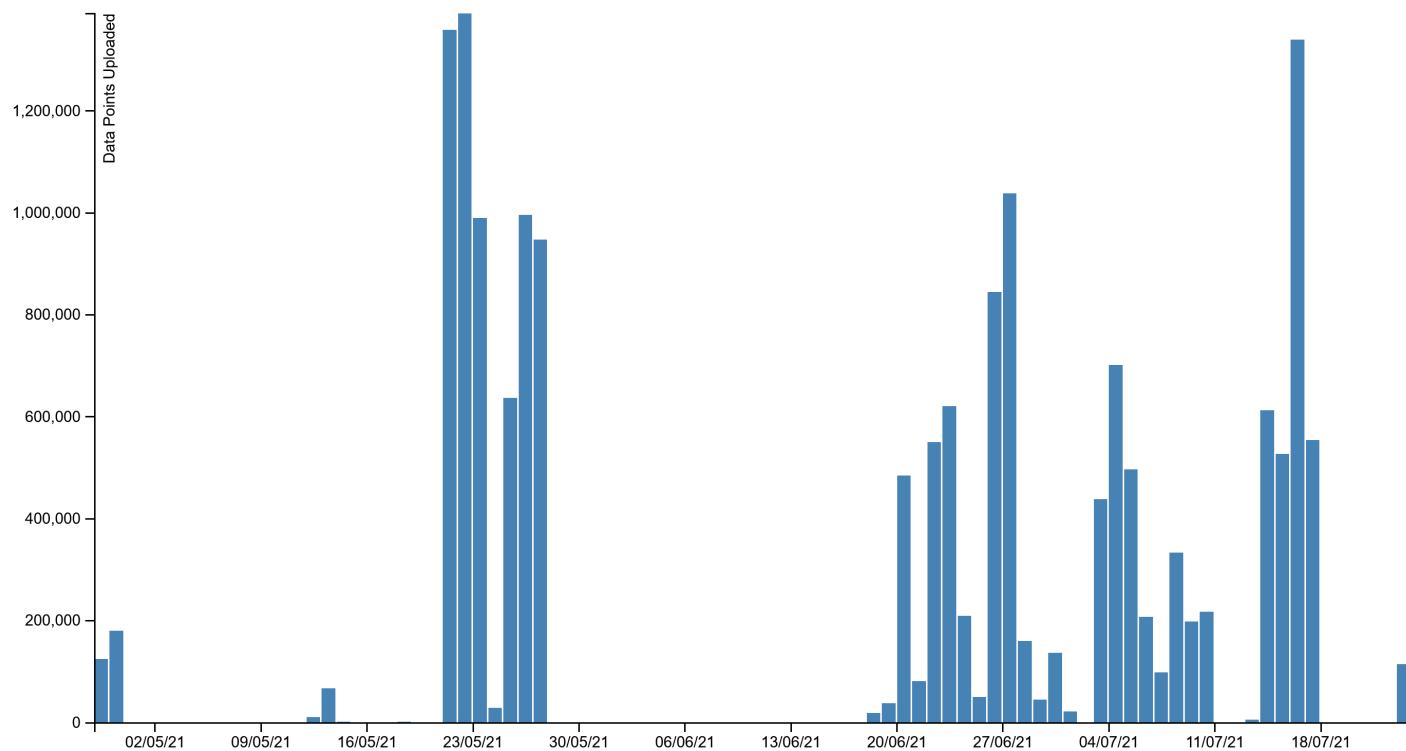
- **Challenge 8 (VERY LIGHT | EXTRACT | ARM):** challenge-8-x86-Linux.exe takes a license key as its first argument on the command line. It is your job to find this key. The program prints an error message if the key is wrong
- **Challenge 9 (MEDIUM | EXTRACT | ARM):** challenge-9-x86-Linux.exe is a protected version of the **bsdtar** program provided by libarchive. A list of command-line arguments can be generated with `./bsdtar --help`. An additional, undocumented argument (`-K "some string"`) can be used to pass a license key to the program. Valid tarballs are produced only when this key is correct. Multiple license keys are possible. It is your job to write a key generator with which correct license keys can be generated. To generate a tarball use this command: `./bsdtar --create --gzip -f tarball.tar.gz -K $KEY $INPUT_DIR`. To list the files in the tarball use this command: `./bsdtar --list -f tarball.tar.gz -K $KEY`. This command will fail if the tarball is corrupted.
- **Challenge 10 (LIGHT | TAMPER | X86)** You are given a binary challenge-10-x86-Linux.exe whose first argument is a password and whose second argument is the input to a hash function. The output is a hash of the second. Your task is to remove the password check from the binary. The cracked binary should take 1 argument and function identically to the original binary if the correct password was entered. Submit the cracked binary.

Figure 4.4: Challenge descriptions for The *Grand Re* round 2.

Challenge	Protection Framework	Platform	Size (KB)
1	Tigress	x86/ARM	131/121
2	Tigress	x86	574
3	Irdeto	x86	35
4	Irdeto	x86	59
5	ASPIRE	ARM	8
6	ASPIRE	ARM	197
7	Tigress	x86/ARM	961/570
8	ASPIRE	ARM	10
9	ASPIRE	ARM	1262
10	Tigress	x86	2775

Table 4.4: The tools used for and sizes of The *Grand Re*'s challenges.

their activity periods as shown in Figure 4.5's histogram of upload activity over time. Moreover, these sign-ups proceeded to produce data and solve challenges. Exceeding *RevEngE CTF*, The *Grand Re* Challenge resulted in the largest dataset collected yet, with over 108 hours of active time, as shown in Table D.1 — just over 58 of which were from the two winning subjects who in total submitted 4 and 3 successful solutions, as announced in Figure 4.2 and listed in 4.9. Following the close of The *Grand Re*, the research team attempted to annotate part of the the traces for the winning subjects.



4.8 Data Collected from the Events

Altogether, 3 individual subjects successfully solved 7 reverse engineering problems, submitting a total of 10 valid solutions to these problems, summarized in Table 4.8. The protection level⁹ of these solved challenges ranged from very light to medium strengths. Subjects solved all three types of challenges — password extraction, deobfuscation and decompilation, and tampering. They also solved problems protected with both Tigress and ASPIRE, but no subjects solved challenges supplied with proprietary Irdeto tools.

Event	Windows	Processes	Screenshots	Mouse Input	Key Input
RevEngE Pilot + CTF	20684	1658428	10496	27200	95103
Grand Re Challenge	66907	16602159	116285	111393	379064
Total	87591	18260587	126781	138593	474167
Size on Disk	42 MiB	14 GiB	10.7 GiB	56 MiB	202 MiB

Table 4.5: The number of data points for each data category for each event, and the size on disk for the tables holding those data categories. The total size for the database is about 25 GiB.

Subjects successfully deployed and ran Catalyst while solving these challenges. In all, these events generated over 130 hours of active trace data to analyze, using about 25 GiB of persistence to store 19,087,719 individual data points. The data is high fidelity, as expected, with no apparent losses in the successful participants' traces; these particular participants used Ubuntu on x86. Other traces, upon review,

⁹"Protection level" refers to the complexity of the obfuscation — the more complex the transform, the heavier the protection level. Complexity here refers to the amount of change between the original program and the obfuscated challenge. Complexity was determined collaboratively by the research team (University of Arizona and Ghent University). Newer versions of Tigress can compute Software Complexity Metrics, which will be useful in future events to automatically determine the difference in source code complexity between original and obfuscated code.

Event	Prizes Offered	Prizes Claimed	Details	Challenges Solved
RevEngE Pilot	\$5,000	\$0	\$100/problem	0
RevEngE CTF	\$7,500	\$1,500	\$500-\$2000/event	3
Grand Re Challenge	\$10,000	\$7,000	\$1000-\$5000	7

Table 4.6: In order to attract qualified subjects for the three events in the study, each featured prize money. The amount of money and rules to win changed for each round as well; the initial RevEngE challenge featured per-problem-solved prizes while the other two offered prize money for winning the competition. In all three events, a lesser amount of prize money was offered for partially complete attempts as well.

Arch	Filtered OS	Count
AMD64	Kali	16
AMD64	Ubuntu	10
Intel64	Debian	1
Intel64	Windows	10
Intel64	Ubuntu	74
Intel64	Kali	9
armv7l	Raspbian	3

Table 4.7: Subjects used a variety of environments and solved both x86 and ARM challenges. Ubuntu, however, was the most common operating system among subjects.

show some issues with other environments, such as Windows deployments stopping recording on their own after some time and Kali window stacks not appearing in the correct order and with strange names.

Subjects earned a total of \$8,500 in prizes across all events in order to generate this data. Given the roughly 130 hours of active trace from subjects who claimed prizes, these events paid roughly \$65 per hour to reverse engineers for their work. Given the participation levels of each competition event, the research team believes that the event advertising played a crucial role.

Figure 4.5 shows the distribution of users' trace uploads over time. Participation distribution appears to be erratic; actions such as starting The Grand Re's second round do not appear to have an impact.

It seems that larger prizes coupled with a larger research team (and thus more perceived validity and connections to other experts) proved invaluable in finally generating a significant number of results. However, other factors also differentiated these events, as summarized in Table 4.9. In particular, a narrower focus, with fewer challenges in the event, could have encouraged participation; event duration and compensation types have an ambiguous effect of participation, though closed studies seem to generate some results while competitions vary according to other variables. All of these variables might be further investigated in future work.

Were these efforts worth it — does the data collected answer research questions about reverse engineering and code protection? The research team has annotated some of the collected data in order to determine the most effective annotation strategies and further developing annotation tooling in the process. Currently, the team is working to annotate more of that data, all of which this dissertation discusses in Sections 5 and 6.

ID	Type	Protection Intensity	Obfuscation Type(s)	Protection Type(s)	Tool	Times Solved	Platform
RevEngE CTF							
1	Password Extraction	Light	Virtualization Opaque Predicates		Tigress	1	x86
2	Password Extraction	Medium	Virtualization Opaque Predicates Self Modifying		Tigress	1	x86
3	Deobfuscation and Decomilation	Light	Virtualization		Tigress	1	x86
Grand Re							
5	Password Extraction	Light	Opaque Predicates Branch Functions Function Flattening	Call Stack Checks Code Guards with Reaction Mechanisms	ASPIRE	2	ARM
7	Tampering	Light	Encode Literals Function Flattening	Checksum	Tigress	1	ARM/x86
8	Password Extraction	Very Light	Opaque Predicates Branch Functions Function Flattening	Call Stack Checks Code Guards with Reaction Mechanisms	ASPIRE	2	ARM
10	Tampering	Light	Opaque Predicates Function Flattening		Tigress	2	x86

Table 4.8: Subjects solved 3 RevEngE challenges and 4 Grand Re challenges; the challenge details are listed above.

Attribute	RevEngE	RevEngE CTF	RevEngE Study	The Grand Re
Team Member Count	2	2	2	6
Format	Large competition	Medium competition events	Closed study	Small competition
Largest Prizes Available	\$100 per problem	$\leq \$2000$ per round	\$500 per problem	\$5,000
Total Prizes Available	\$5,000	\$7,500	\$4,000	\$10,000
Duration	Roughly 1 year	7 to 14 days per round	Up to 6 days per problem	59 days
Successful Submissions	0	0	3	7
Challenges Generated	80	82	4	10
Compensation Type	On problem success	On round win	On problem attempt success	On attempt and event win

Table 4.9: Study event details and differences are listed here. Events are ordered by their start date, increasing from left to right; Catalyst maturity also, then, increases left to right. Note that "attempt" as used here refers to smaller amounts of compensation for unsuccessful submissions with valid traces showing a good faith solution attempt.

4.9 Ghent University Student Study

Following the RevEngE and Grand Re events, in December 2021 researchers at Ghent University employed Catalyst to conduct the framework’s first external study. Specifically, the Ghent University members of the research team hosted a reverse engineering challenge event for their students. This study proved fruitful: Students generated over 4600 minutes of reverse engineering trace with Catalyst. Moreover, the Ghent team deployed a copy of the Catalyst framework on their own server, the first such deployment outside of the University of Arizona.

The instructions, shown in Appendix J, required that the students reverse engineer two problems while running Catalyst. Specifically, students were required to extract keys from two binaries. All students received the same first binary, which included static opaque predicates. The second binary included dynamic opaque predicates but otherwise varied across three student groups — specifically, different groups received binaries using different data structures — in order to study the learning effect of the first challenge.

A total of 19 students participated in this study out of 38 who signed up and signed a consent form. Of those 19, 14 were able to download, install, and run the Catalyst. Of those 14, 9 produced excellent quality, likely useful traces as determined by the Ghent team upon review. All of this produced a total of 4120 active trace minutes (approximately 69 hours) with specific results summarized in Appendix L. The students also submitted surveys asking them to detail what they did during the study. Analysis of the traces, surveys, and submissions determined that students used GDB and Ida; all but 2 participants solved the first challenge but only 1 participant solved the second challenge.

4.10 Summary

Catalyst gathers HDI data from users as they use their devices; the research presented in this dissertation aims to gather HDI data from software reverse engineers specifically. The initial research team determined that using reverse engineering events (which give participants reverse engineering challenges to solve) in conjunction with Catalyst could be an effective method to generate the desired data. Thus, we explored the types of reverse engineering challenges that parallel real-world software reverse engineering practices, as discussed in Section 4.2 and integrated these challenge types into a reverse engineering problem generation framework — RevEngE. We then configured the system and announced the *RevEngE Pilot* event using recruitments described in Section 4.4. This initial pilot event demonstrated that our basic tooling worked, but failed to attract participation, as discussed in Section 4.5. Addressing a set of potential shortcomings, the initial research team then developed a successor event, *RevEngE CTF*, which ultimately generated a significant amount of high quality data as shown in Section 4.6. In all, these events collected many hours of the high-fidelity reverse engineering HDI data, specified in Section 4.8. Expanding on this effort, the initial research team collaborated with researchers and practitioners from Ghent University and Irdeto in developing and deploying the *Grand Re*; this event produced the greatest amount of data with a larger variety of challenges, as shown in Section 4.7.

Chapter 5

TOOLS FOR VISUALIZATION AND ANALYSIS

Once data has been recorded from the endpoints and made available from the Catalyst back end, it is ready to be reviewed and resolved into a model (an attack Petri net). This is accomplished by visualization portion of Catalyst.

An overview of the analysis and modeling methodology is shown in Figure 5.1: First, analysts select sessions to work on (each subject may have worked over multiple days on a particular challenge, and each work period is recorded into a separate session), then view those sessions and annotate the low level, highly granular human-device interaction data by describing what the subject did during a particular time period, and, finally, those annotations are processed automatically into Petri nets.

Henceforth, an **annotation** is a time period label with a start and an end time. It represents the set of the collected time-series datapoints that lie between the start and end time, such as the keystrokes a subject types and the screenshots captured during the time period (see Figure 3.1). Human analysts generate these annotations using the Catalyst visualization tool, though future work may introduce automated or semi-automated methods to help generate them.

In addition to the start and end times, analysts give an annotation contain a label (the “Task Name”), a Goal, a set of Tags, a Note, and a Completion Metric. A Task Name contains the overall annotation’s task name — the thing a subject is doing; a Goal describes what the subject was trying to achieve during the time period; a Tag, similar to tagging concepts in social media, a categorical data point that can be used to convey common information like tools or processes, which can

be used for information retrieval and comparison later; a Note is a freeform, high-level understanding of an annotation which helps convey any information missing from other annotation attributes; and the Completion Metric describes how well the subject completed their Goal — it is a float which, in the current user interface, ranges from 0 to 100, where 100 would indicate full completion and 0 indicates no completion.

This chapter first provides an overview of the analysis pipeline in Section 5.1, details the target environment and its performance constraints in Section 5.2, explains the annotation visualization views in Section 5.3, and explains the annotation to Petri net generation algorithm in Section 5.4.

5.1 Visualization and Analysis Goals

In the dataset, annotations act as labels for other time series data: They describe what a subject does during a particular time period. The data points in between the start and end are what a user does with a device to conduct the task described by the annotation; annotations are labeled based on the underlying data and the underlying data offers a more granular view of what the annotation describes. With that in mind, the aim of this research is to generate human-understandable models of subjects' behavior to accomplish these goals:

- Enable a human (who is familiar with the subject matter) to replicate what a subject did and understand the methods a subject used.
- Determine which subjects' methods are most and least effective.
- Gauge the resources needed to perform different attacks.
- Compare subjects' methods to understand which are common.

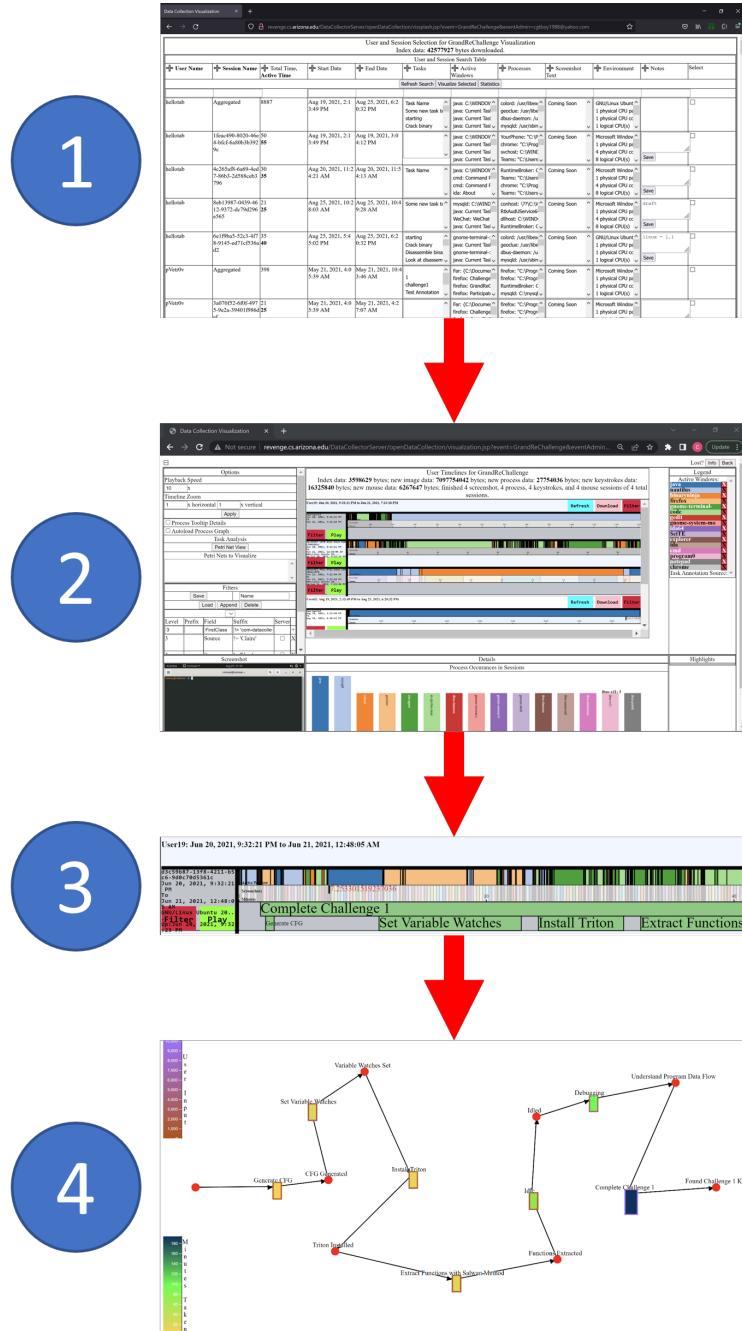


Figure 5.1: The overall visualization and analysis process: (1) Select sessions to view on the session selection table; (2) visualize those sessions; (3) annotate the sessions in the visualization; (4) generate Petri nets from those annotations.

In order to meet these goals, the visualization must enable a group of collaborating researchers to produce high quality annotations from the collected traces. Once generated, analysis methods transform the annotations and underlying data into models to accomplish the research goals above.

The visualization, shown in Figure 5.2, contains several views which focus on understanding the subject's method, entering annotation data precisely, and viewing models built from the annotations. The timeline and animation views provide the primary method of both understanding the method and entering annotation data. The session selection table, shown in point (1) of Figure 5.1, allows analysts to select to work on specific sessions.

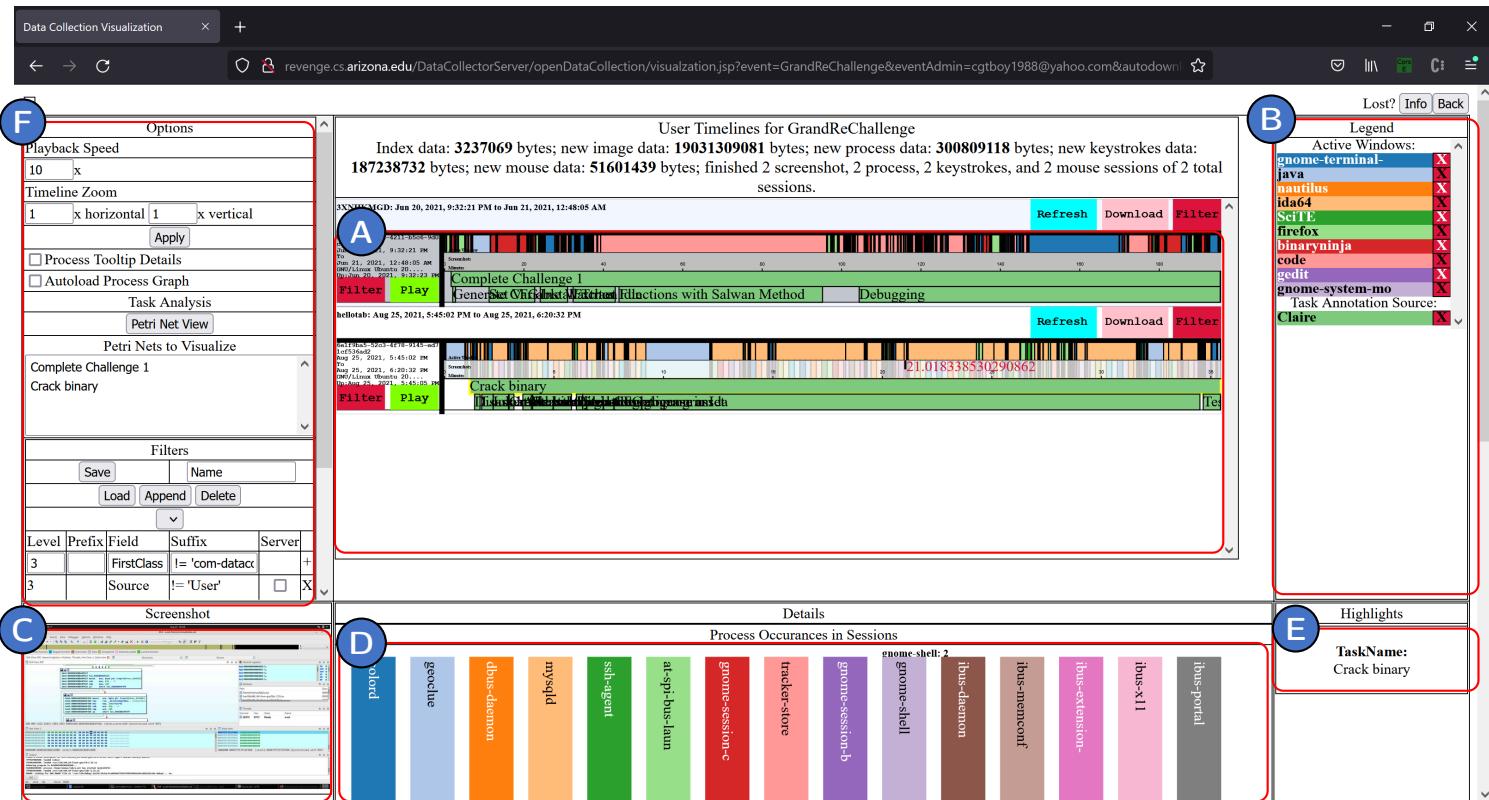


Figure 5.2: The overall visualization with two sessions loaded in the default timeline view, consisting of (A) the timelines, (B) the timeline legend, (C) screenshot preview, (D) all-sessions summary data, (E) data highlights, and (F) the options. Not in the screenshot are the process graph, data details table, animation view, and Petri net visualization.

5.2 Loading and Environment Considerations

Collaborators on this research include individuals on multiple continents and with a variety of computational resources and environments. We also anticipate this system to be used for future computer security studies, not just reverse engineering, connecting researchers from around the world to collaboratively collect, annotate, and analyze attack data. In order to make the visualization tools maximally accessible to collaborators and the public when a dataset is published, the visualization was implemented to run in a web browser. This ensures that the visualization is easy to access and use, annotations are consistent across sites, and the system requires practically no environmental setup prior to use. However, the web browser environment introduces numerous constraints which must be taken into account.

Currently, the sum of all data sets collected requires about 25 GiB of persistent storage. It is not possible to quickly transfer 25 GiB of data over commodity connections, nor is it possible to store that much data in browser memory. Instead of attempting this, the browser first loads a summary table to select sessions to visualize. Each session is only a small fraction of the overall dataset, with the size of each session varying according to environment, activity, and length.

Once selected, the visualization page loads the data for the sessions. Loading a large, monolithic data export and parsing the encoding requires significant memory resources on both the client and server, as well as connections that have very little interruption. This strategy does not work on the larger sessions in the current dataset and causing browser crashes. Instead, the data is served in pieces and stored on local browser persistence, as shown in Figure 5.3.¹ Connections are throttled and limited in order to reduce browser and server memory use, and large data chunks (such as

¹IndexedDB is a database engine built into modern browsers [294]. It is currently the only way to access the amount of persistent storage required by the visualization in a browser.

screenshots and process data) are left in persistence until needed, further reducing memory footprint.

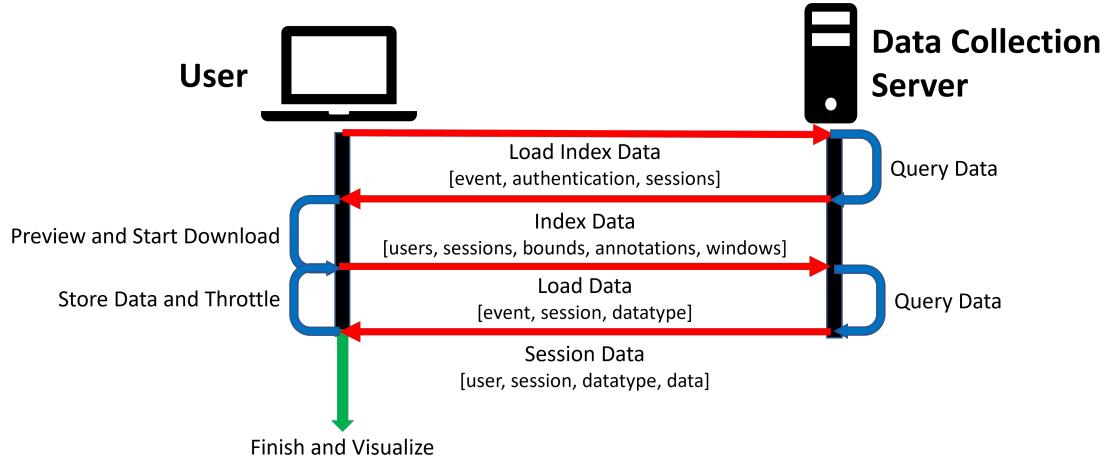


Figure 5.3: The visualization begins with loading and showing small-sized index data before using multiple, asynchronous, throttled requests to load and store the rest of the data. Finally, when all of the data has been loaded, the entire dataset for the selected sessions is visualized.

The resulting visualization system thus keeps a significant amount of data out of memory, but accessing that data requires waiting for persistence to return it, and that latency must be taken into account when generating visual elements. While none of the operations are computationally complex², with none exceeding $O(n \log n)$ complexity, some of the data components are quite large — namely, the process data and screenshot data. Bottlenecks in various components are noted as the components are discussed.

²An exception is the force directed Petri net graph visualization, which operates in $O(n^3)$ on the small annotation section of the dataset.

5.3 Visualization User Interface

The **timeline view** of the visualization (shown in Figure 5.4) provides a broad view of a session. Pane (A) shows the username; pane (B) shows the session name, along with buttons to filter out the session and play/resume the session’s animation; pane (C) shows the current active window; and pane (D) includes a horizontal axis showing the session time scale in minutes; pane (E) displays the annotations along the timeline. Clicking a window in (C) causes the details table (at the very bottom middle of the visualization) to present information about the active window. (D) highlights the exact time at a given point when the mouse is over it, and clicking on it starts the animation view at that time point. The rectangle’s color matches an entry in the legend in Figure 5.2 (B).



Figure 5.4: The timeline view presents each sessions’ windows, screenshots, and annotations on a time scale axis.

Pane (D) represents individual screenshots captured. Placing the mouse cursor over this shows the closest screenshot for that given point in time in the lower left corner. The cursor also shows the exact time into the session the point represents as it scrolls along the bar. Clicking on this bar starts the animation view playing at that given point in time.

Pane (E) in Figure 5.4 shows the annotations themselves, arranged in a condensed Gantt chart format, where each task/annotation is placed in the nearest open slot rather than given its own line [220]. This is done to reduce vertical space and

allow more sessions and annotations to be shown concurrently without scrolling. The annotations are labels for everything that happens during the time period they cover. As with windows, clicking annotations displays its details and allows fields to be edited.

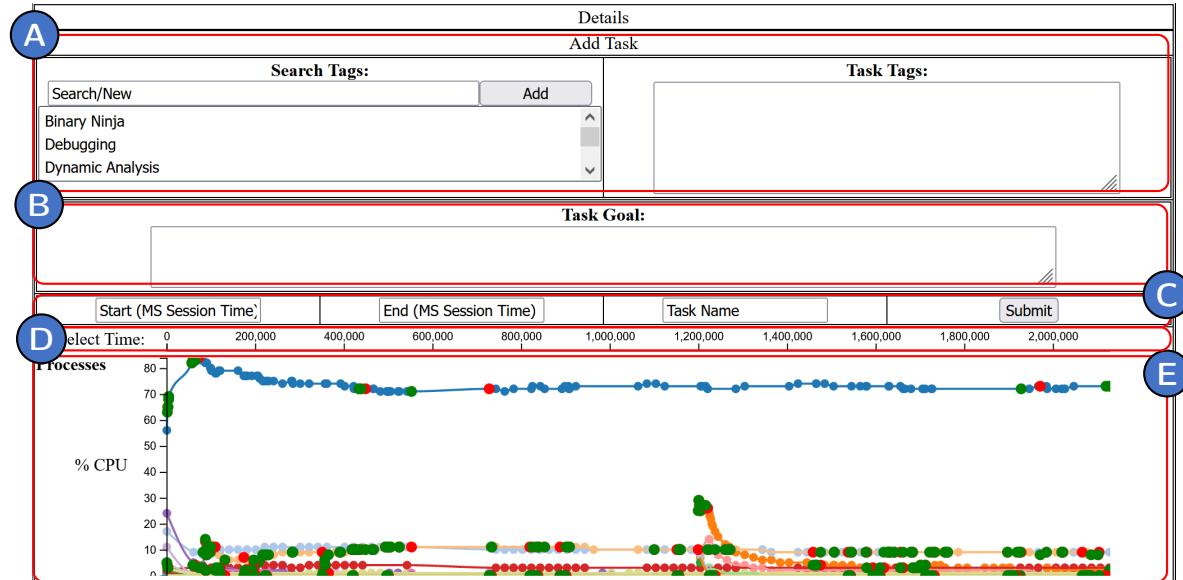


Figure 5.5: The process graph right after loading and showing time selected and mouseover tooltip. Green circles indicate a process’s UI becoming the active window, and red circles indicate it exiting the active window status.

Panes (A), (B), (C), and (D) in the *process graph view* for the session (shown in Figures 5.5 and 5.6) allow the analyst to enter annotation data for a given time period. This includes a number of “tags” (from a predetermined list of common tasks) and “goal” (the perceived goal the subject was working towards in the current activity). (D) is a brushable time axis to select the start and end time for an annotation. Pane (E) displays resource usage such as CPU and memory. Each process is given a color corresponding to the legend in Figure 5.2 (B).

The **animation view**, shown in Figure 5.7, displays what a user does in a session

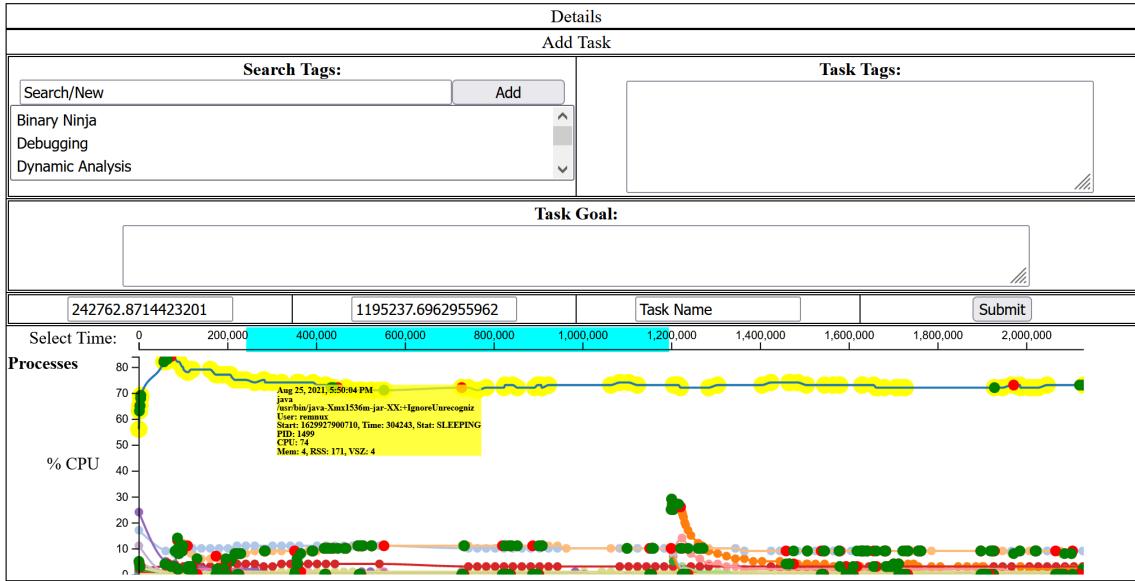


Figure 5.6: The same process graph in 5.5 showing a time selection on the timeline and a mouse over a process data point.

like a video. The animation view consists of the current screenshot (A) with a selectable and seekable timeline below. (B) shows the keyboard input, (C) the top 5 processes by CPU load, and (D) contains play/pause buttons, annotation seek, the task menu toggle, and current active annotations. The fast forward button will seek to the next annotation event — either an annotation starting or ending

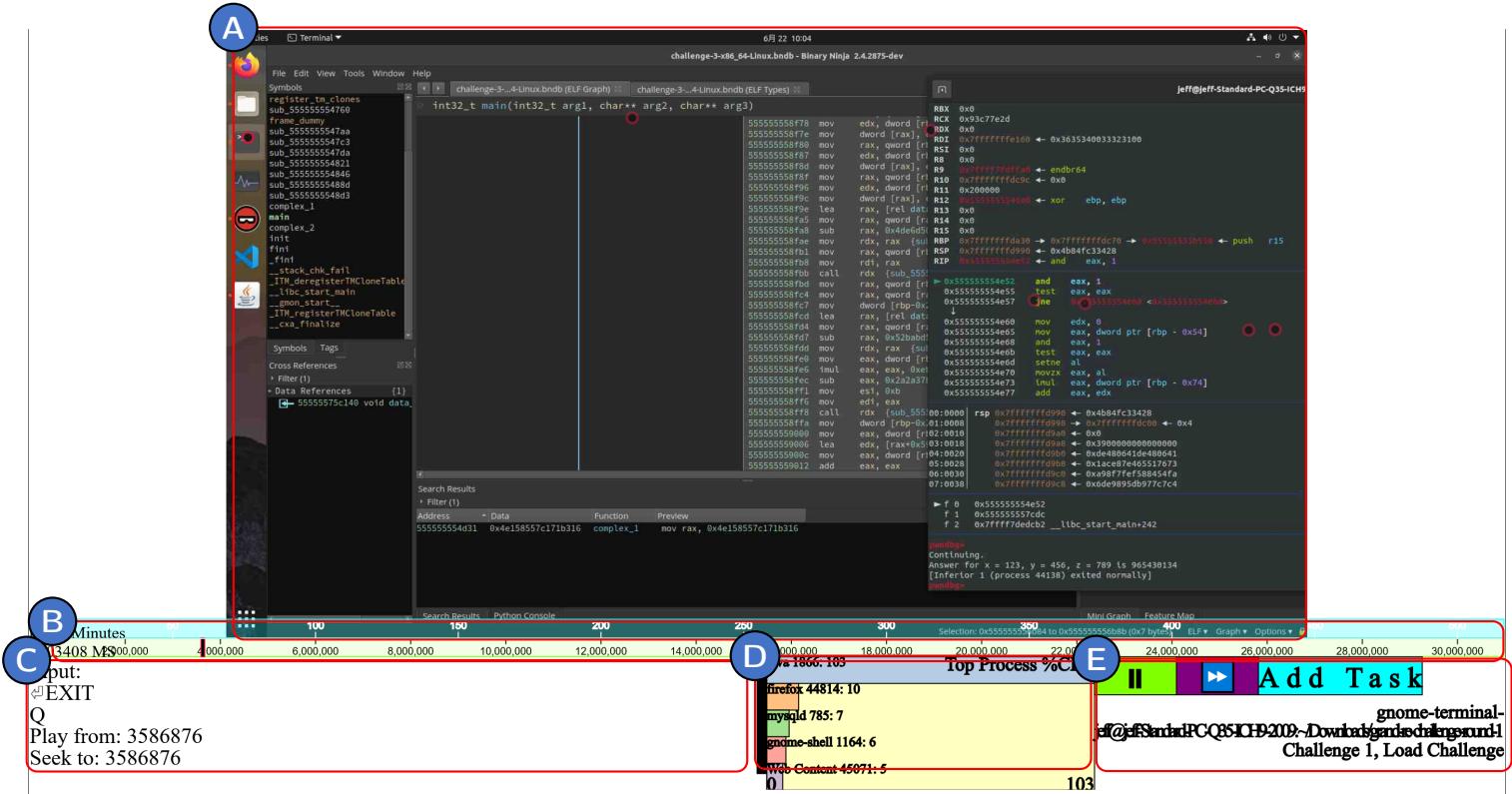


Figure 5.7: The animation view: (A) plays a video of screenshots with mouse clicks superimposed; (B) contains the timeline with clickable seeking and brushable time selection for annotation entry, (C) contains text input; (D) has the top 5 CPU use processes; and (E) contains play/pause buttons, a toggle for the task/annotation creation menu, and active annotations and window information.

The **data filtering pane** (Figure 5.2 (F)) allows analysts reviewing data in the visualization to remove certain entries (based on users, sessions, windows, annotation sources, etc.) from view. Sets of filters can be saved and loaded, as many filters can be applied usefully when viewing different sets of sessions in the visualization.

5.4 Petri Net Generation and View

As discussed in Section 2.5.3, Petri nets are a well established way to model complex, dynamic systems. They have been shown to be effective for modeling attacks in computer security. In this section we will explore how the {Task, Goal} tuple annotations produced by analysts enable relatively automated Petri net generation. Moreover, given that the annotations label time periods for which there exists raw data, that data can provide additional input to the Petri net models.

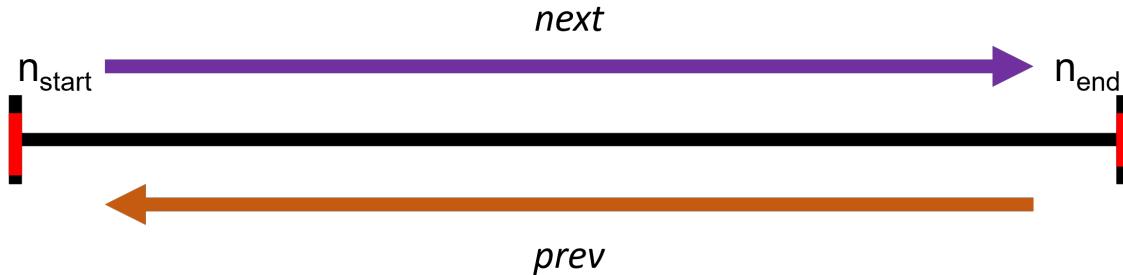


Figure 5.8: An annotation, as pertains to the algorithms described, consists of a start and end node on a timeline.

Algorithm Definitions An **annotation** consists primarily of a {Task, Goal} but also contains several other data points including a *note* and 0 or more *tags*, as well as a *completion metric*. In some figures, annotation blocks are labeled as *tasks* to demonstrate what a user is doing in the time period labeled by an annotation; these tasks/annotations are denoted by elements T . Importantly, each **annotation** is split

into two **nodes**, one of which is a **start node** — n_{start} — and one of which is an **end node** — n_{end} — as illustrated in Figure 5.8. Each node has a *timestamp* and each annotation node pairs are linked to each other (with entries *next* and *prev*) and the other annotation data fields. In the data considered here, annotations are always represented as their node pairs. Sets of annotation datapoints (the sorted sets D in described algorithms) consist of sorted annotation nodes — both start nodes and end nodes for a given set of annotations are in these lists unless otherwise noted by the described algorithm. In those sets, nodes are labeled a ; individual nodes considered by the algorithm are denoted by n .

5.4.1 Annotations and Petri Nets

Petri nets consist of states (represented by rectangles/bars) and transitions (represented by circles). A transition is an action, the outcome of which changes the state. Annotations label what users do — their actions — and thus can be considered transitions in the Petri net. The goals of these annotations (if successful) change the subject’s state by adding what the subject accomplished to their current state and thus are added to states in the Petri net, as illustrated in Figure 5.9.

5.4.2 Annotation Relationship Types

Annotations, however, are not necessarily single-level, linear matters. Complex tasks consist of multiple subtasks, which themselves may have subtasks. Some tasks lead to other tasks, and subjects might attempt to do multiple tasks at the same time. The relationships between annotations suggest the order in which subjects must complete tasks to execute their methods, and are thus important in generating models generally and Petri nets specifically.

Here, it is assumed that subjects complete tasks in monolithic blocks — that is,

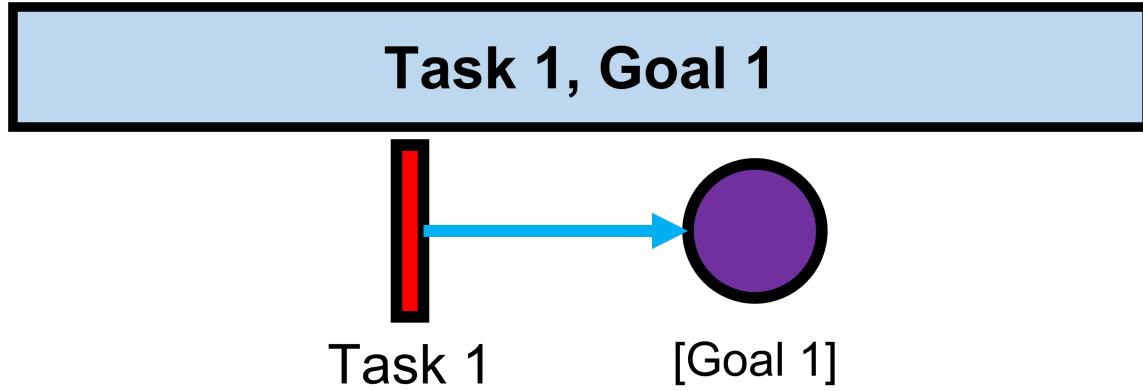


Figure 5.9: Given an annotation with a {Task, Goal} tuple, a corresponding Petri net consists of a Transition and a State; the Task, describing the action, becomes the Transition, while the Goal, being the outcome, is added to the State.

each task has a concrete start and end point. In practice, this is not necessarily the case; subjects might start a task, pause that task, and resume it at a later point. Where this occurs in the dataset, it is represented by breaking the annotation into multiple pieces, each covering a portion of time where a subject engages in the task. Where these breaks are so small or ambiguous that they are impossible for annotators to discern, the annotation is instead the entire block of time without any breaks.

Given the problem constraints, annotations here have relationships typical to tasks in Gantt workflow charts, as suggested by the Gantt chart visualization used to show annotations in Figure 5.4 (E). Those relationships, and the connections they are assumed to imply here, are as follows:

hierarchical: An annotation can completely encompass another, where the first is the “parent” to the second, which is a “child” (Figure 5.10). The child is considered a necessary component of the parent.

concurrent: An annotation can overlap with another, where there are sections of

both concurrently (Figure 5.11). This occurs when a subject is multitasking.

sequential: A sequential annotation can have no overlap with another (Figure 5.12).

The first annotation must complete before the second can execute.

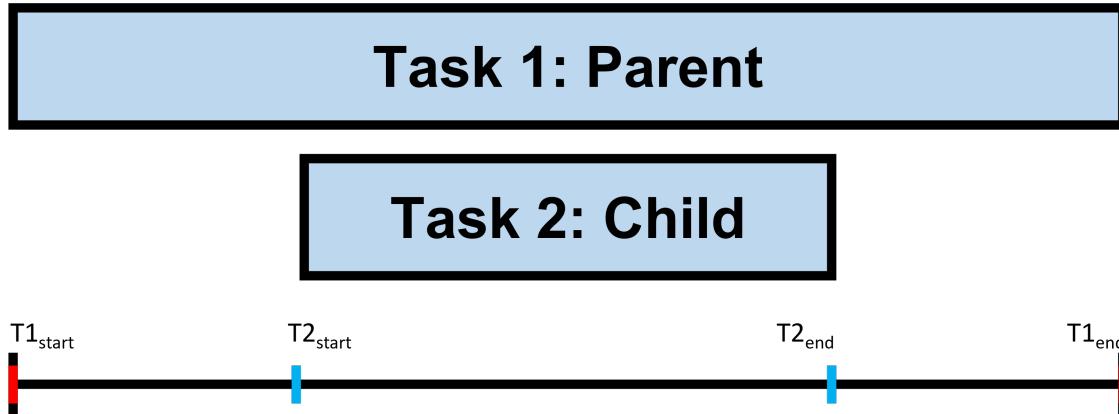


Figure 5.10: A task with a start and end point lying before and after another has a demonstrates a hierarchical relationship, where the longer task is a parent and the shorter a child.

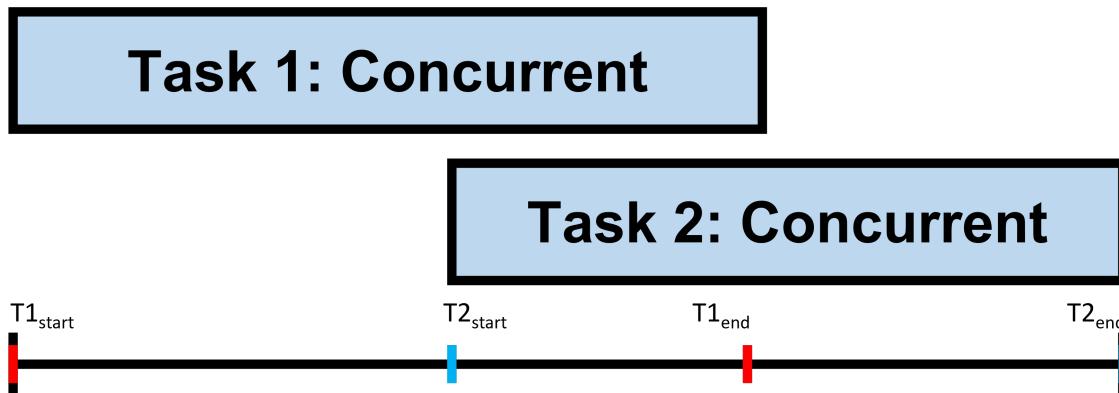


Figure 5.11: Tasks with either a start or end point — but not both — in between the start and end points of another task are considered concurrent to the other task.

These rules are general and assumed to be the case in generating models from annotations. However, there will be exceptions. It is important, then, to enable model

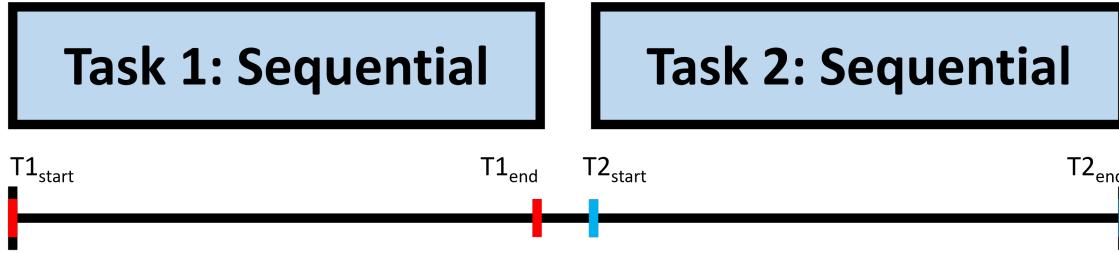


Figure 5.12: Tasks where both the start and end points come before or after the start and end points of another task are considered sequential.

editing to manually correct relationships inaccurately assumed; in future work, it may also be possible to automatically infer exceptions as well. Examples of exceptions to the inferred relationships include:

- We might infer a hierarchical relationship where the child task is assumed to be required to complete the parent though, on closer inspection, the child might actually be done independently. This could be the case when a subject multitasks and this situation should be better understood as a concurrent relationship.
- We might infer a concurrent relationships though, on closer inspection, one task actually has to start (but not finish) before the subject begins a second task. Such a case should be better understood as a type of sequential relationship.
- We might infer a sequential relationships though, on closer inspection, one task may not be required in order to start or end a subsequent task, even though a subject does the initial task first. Such a case should be better understood as concurrent.

In order to best work with the Petri net generation method below, annotators should be aware of these exceptions to the standard rule.

5.4.3 Generating Petri Nets from Annotations

Given the relationships described, it is possible to deduce the actions, order thereof, and resulting states a user executes and attains to complete a task. These, then, can be translated into Petri nets by tracing these relationships into edges between different {Task, Goal} annotation tuples. Expanding on the rules above, we get:

- A parent task generally requires all children to complete before it is completed.
- Descendants beyond direct children have to be completed before children are completed.
- Where a task has multiple concurrent parents, it is most likely associated with the one that began last.
- Sequential tasks generally require the previous task in the sequence to complete before they can start; the previous task state is required to do the next task.
- No causal relationship can be generally inferred regarding concurrent tasks.

Applying these rules to annotations, given their relationships with other annotations, yields an ordering between annotations, generating graph edges between them. Applying the {Task, Goal} separation into Petri net transitions and states as shown in Figure 5.9 translates each annotation into a small Petri net tuple. These tuples are then connected according to the ordering/edges between annotations, with the output state of each annotation leading to the transition of another until the end state of the annotations is reached. Because states are the culmination of the outcomes of all prior actions, the output state for each annotation consists of that annotation's result (goal) as well as the sum of all previous output states. Tracing through anno-

tations assuming these rules hold true yields the example Petri nets shown in Figures 5.13, 5.14, 5.15, and 5.16.

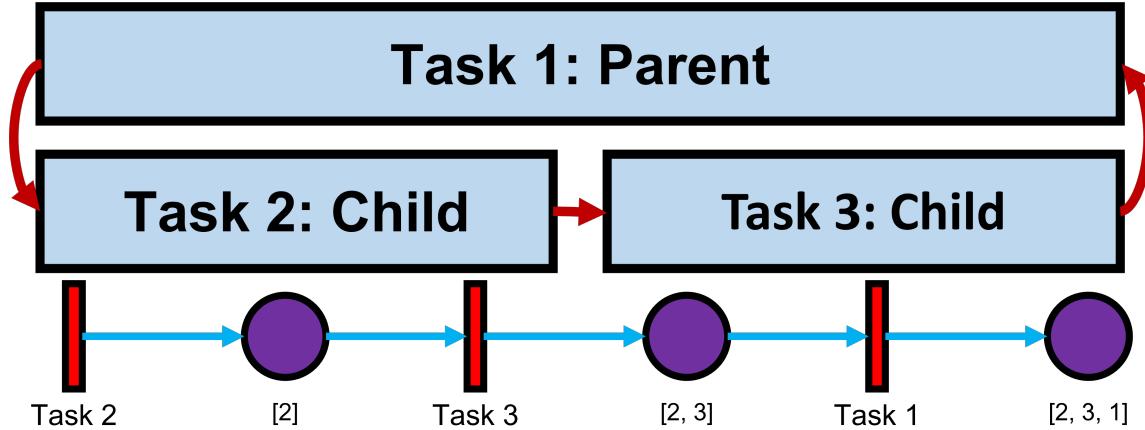


Figure 5.13: Task 2 is required to start and finish Task 3; the steps to complete Task 1 are Task 2 followed by Task 3.

5.4.4 Petri Net Generation Algorithm and Performance

Applying the rules listed in Section 5.4.3 to annotations generates a Petri net. A $O(n^2)$ algorithm to apply those rules to annotations is described here. In practice, $O(n^2)$ is sufficiently performant, as annotation datasets tend to be small and (see 4.8) have characteristics that cause the presented algorithm to perform close to linearly. Broadly, this algorithm consists of 4 main steps:

1. Detect hierarchical and concurrent relationships in annotations (Algorithm 4)
2. Detect sequential relationships in annotations (Algorithm 5)
3. Calculate the cumulative state of each annotation (Algorithm 6)
4. Convert the resulting relationships into a Petri net graph consisting of transitions, states, and edges (Algorithm 7)

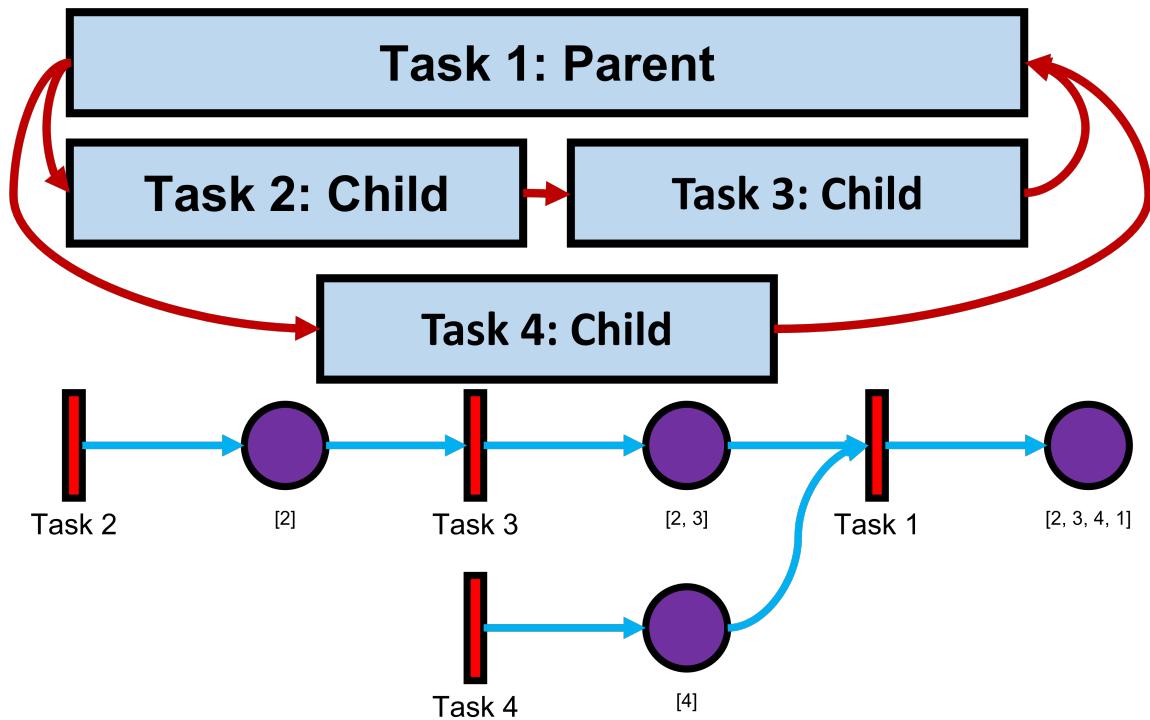


Figure 5.14: Same as Figure 5.13, except Task 4 is also required to complete Task 1 but has no other relationship to Tasks 2 or 3; it is being done concurrently and independently.

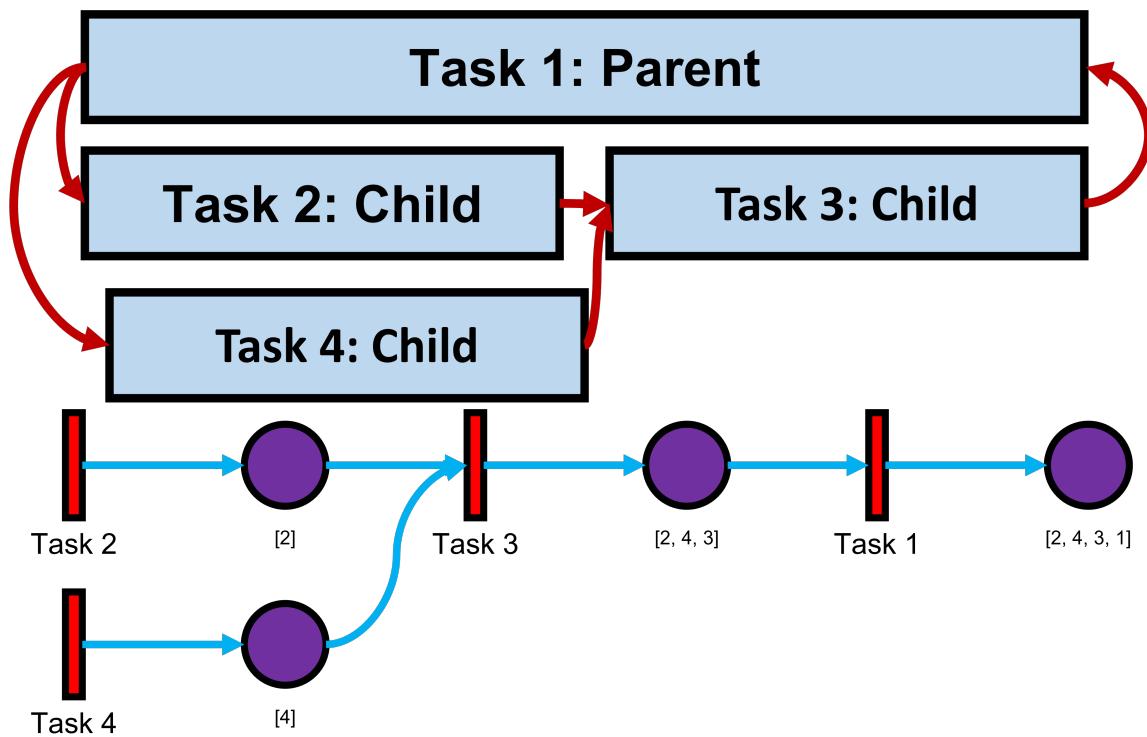


Figure 5.15: Both Task 2 and Task 4 are predecessors of Task 3, but are independent of each other due to concurrency.

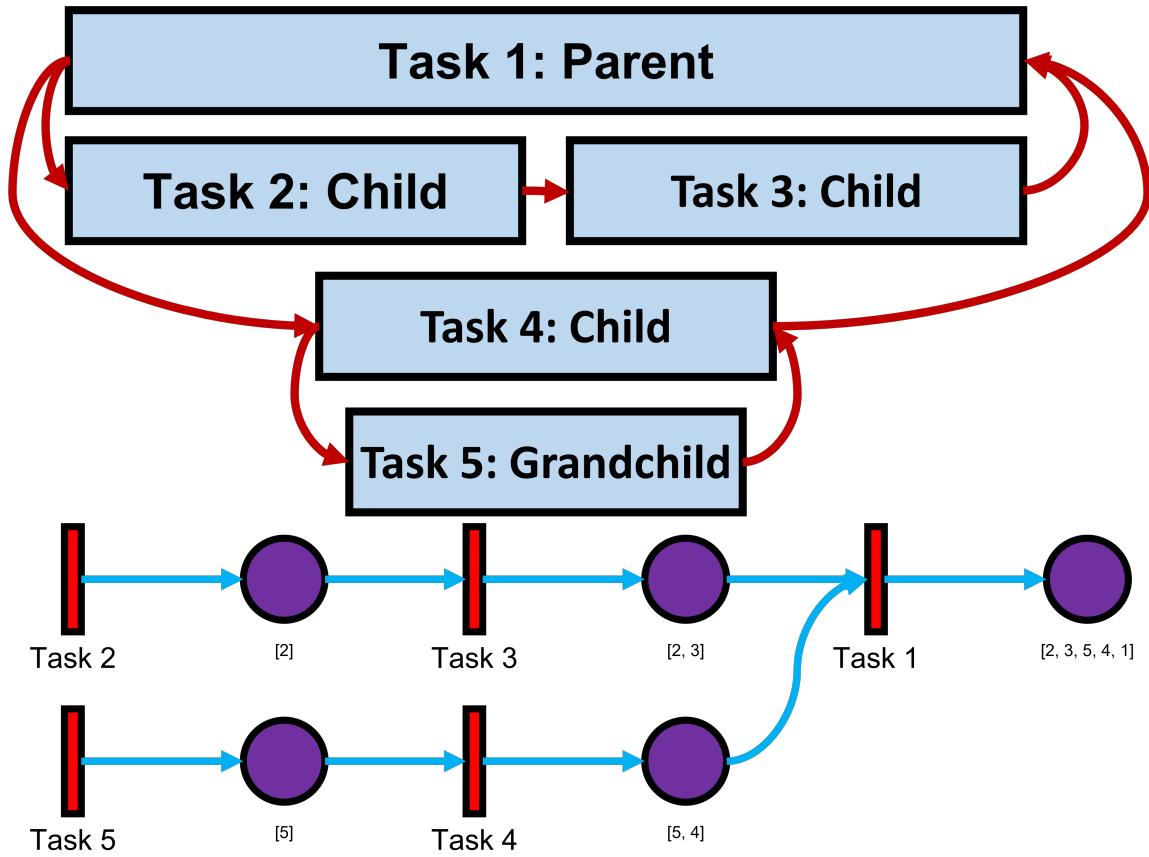


Figure 5.16: Same as Figure 5.14 but demonstrating the behavior of descendants with a grandchild annotation. Note that the grandchild is required by its direct parent, which in turn is required by the grandparent. This result is the same even if Task 5 was short enough to end before Task 2, since its last-starting parent is Task 4.

Each annotation consists of two nodes: a start node and an end node. Each node has a timestamp, and they are linked together by reference. Nodes from all annotations are provided in a list sorted by timestamps called $D_{annotations}$, consisting of a tuple with these attributes:

$$\begin{aligned} a \leftarrow & \{ Time, Task, Goal, Tags, Note, Start \mid End \} \\ \leftarrow & \{ Children, Concurrent, Parent, Next \mid Prev \} \end{aligned}$$

These attributes serve to provide an abstracted understanding of what a subject is doing during the time period. Note that $Start \mid End$ is an enumeration determining whether a node is n_{start} or n_{end} and $Next \mid Prev$ is a pointer to either n_{end} from n_{start} in the case of $Next$ or vice versa in the case of $Prev$. $Children$ and $Concurrent$ are initially empty sets that the algorithms described here fill with nodes with their namesake relationship described above; similarly, $Parent$ is an empty pointer until filled by the algorithm.

To calculate the different types of relationships between annotations, the algorithm starts at an index in the annotation list, either the first element in the list if generating Petri nets for all annotations or with a binary search to a selected annotation to generate for just that annotation. Arguments are specified accordingly — 0 and ‘all’ or the binary search result and ‘single’.

The algorithm for detecting hierarchical and concurrent relationships (Algorithm 4) begins by iterating over all relevant annotations depending on the arguments supplied. For each iteration:

- If n_{next} is also n_{end} then the algorithm reached the end of the current annotation:
 - If the mode is ‘single’ then end, since the algorithm in this mode stops calculation at the end of the current annotation. Return the iterator.

- If the mode is ‘all’ then increment the iterator, reassign n_{start} and n_{end} , and continue the Loop to process more annotations.
- If n_{next} is an ‘End’ node and the corresponding ‘Start’ node’s timestamp is before n_{start} ’s timestamp, then add that annotation node pair to n_{start} ’s list of concurrent nodes, because the ‘Start’ node is before the current annotation but the ‘End’ node is during the current annotation.
- If n_{next} is a ‘Start’ node and its corresponding ‘End’ node timestamp is after the n_{end} timestamp, then add that annotation node pair to n_{start} ’s list of concurrent nodes, because the ‘End’ node is after the current annotation but the ‘Start’ node is during the current annotation.
- If n_{next} is a ‘Start’ node and its corresponding ‘End’ node timestamp is before the n_{end} timestamp, then the n_{end} lies completely within the current task, and therefore is a child of the current task. Add this node pair to the n_{start} ’s list of children. Call the algorithm recursively, passing the iterator and the ‘single’ mode and assigning it to the return of the call; this calculates the relationships for the children’s children and later descendants as the recursive call goes deeper. A parent annotation lies concurrent to its childrens’ concurrent annotations except those that are also children, so n_{start} adds $n_{next}[Concurrent]$ to $n_{start}[Concurrent]$ except those in $n_{start}[Children]$ after the recursive call calculates n_{next} relationships.
- If n_{next} is an ‘End’ node and the corresponding ‘Start’ node is within the current annotation, it should have already been added to this annotation’s children.

This algorithm iterates once over $O(n)$ nodes (n is the number of nodes — exactly double the number of annotations) and results in each annotation holding an

Algorithm 4 The hierarchical and concurrent relationship detection algorithm.

//(1) The algorithm begins with a list of annotation nodes sorted according to their timestamps.

$$D_{annotation} \leftarrow \{ a_1, a_2, a_3, \dots, a_x \} \mid a_x[Time] < a_{x+1}[Time]$$

//(2) The *SetAnnotationRelationships* function calculates all hierarchical and concurrent relationships between annotations. The argument *i* is an index in $D_{annotation}$ which references the annotation node to process. The *mode* argument determines whether to process just the annotation node at *i* (if it is ‘single’) or process all annotation nodes in $D_{annotation}$ (if it is ‘all’).

procedure *SetAnnotationRelationships*(*i, mode*)

//The selected annotation consists of the start and end nodes n_{start} and n_{end} . The algorithm generates a Petri net by analyzing this annotation and annotations between its start and end nodes.

```

nstart  $\leftarrow D_{annotation}[i]$ 
nend  $\leftarrow n_{start}[Next]$ 
    ///(3) Iterate over all annotations between the start and end of the selected annotation.
    while EXISTS  $D_{annotation}[i + 1]$  do
        i ++
        nnext  $\leftarrow D_{annotation}[i]$ 
        ///(4) The end of the selected annotation is reached when the iterating node (next node) is the same as the end node for the selected annotation.
        if  $n_{next} = n_{end}$  then
            if mode = single then
                //The end node for the selected annotation was reached and the mode is ‘single’ so the algorithm terminates.
                return i
            end if
    end while

```

```

if mode = all then
    //The end node for the selected annotation was reached and the
    mode is 'all' so the algorithm moves to the next annotation (and sets that as the
    selected annotation) if it exists or terminates if it does not.
    if EXISTS  $D_{annotation}[i + 1]$  then
         $i++$ 
         $n_{start} \leftarrow D_{annotation}[i]$ 
         $n_{end} \leftarrow n_{start}[next]$ 
    else
        return  $i$ 
    end if
    end if
    //5) If the algorithm has not yet finished, then it can determine the
    current iteration's node to the selected annotation.
    else if 'Start'  $\in n_{next}$  AND  $n_{next}[Next][Time] > n_{end}[Time]$  then
        //The next node starts during the selected annotation but ends after,
        so it is concurrent.
         $n_{start}[Concurrent] ADD n_{next}$ 
    else if 'End'  $\in n_{next}$  AND  $n_{next}[Prev][Time] < n_{start}[Time]$  then
        //The next node ends during the current annotation but starts before,
        so it is concurrent.
         $n_{start}[Concurrent] ADD n_{next}[Prev]$ 
    else if 'Start'  $\in n_{next}$  AND  $n_{next}[Next][Time] < n_{end}[Time]$  then
        //The next node starts after and ends before the selected, so it is a child
        node.
         $n_{start}[Children] ADD n_{next}$ 
         $n_{next}[Parent] \leftarrow n_{start}$ 
        //Calculate all relationships for the next node and add those results to
        the selected annotation.
         $i \leftarrow SetAnnotationRelationships i, single$ 
         $n_{start}[Concurrent] \leftarrow n_{start}[Concurrent] \cup (n_{next}[Concurrent] \cap$ 
         $n_{start}[Children]')$ 
    end if
end while
end procedure

```

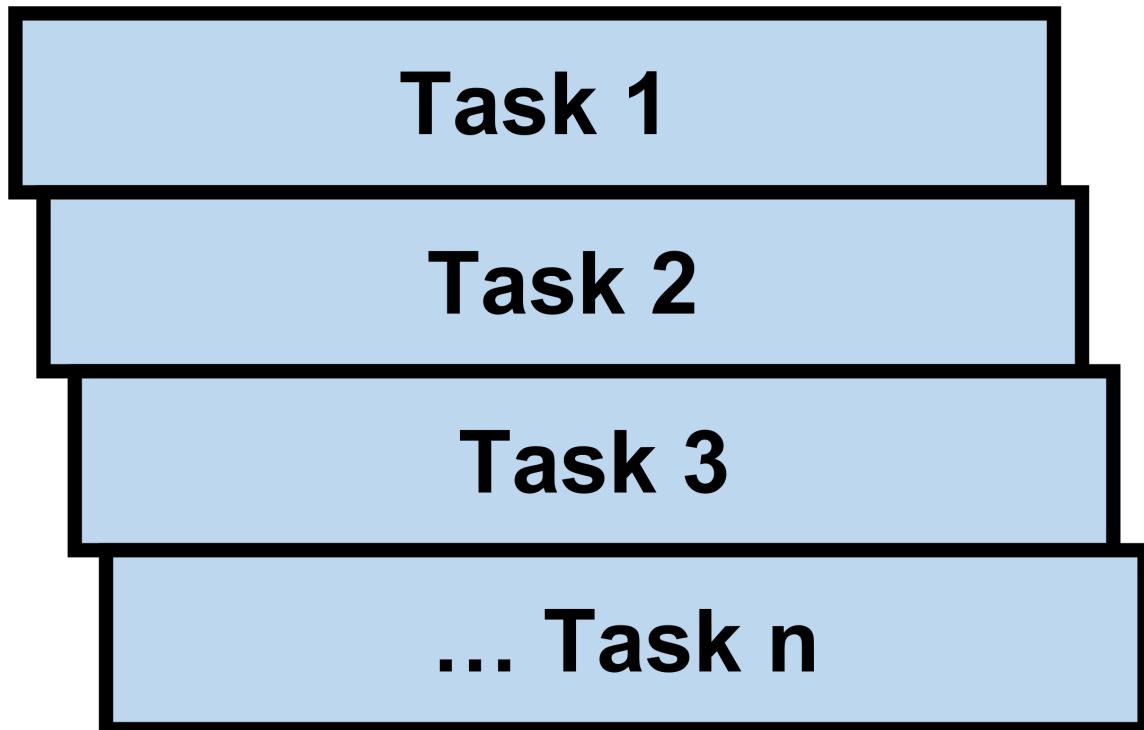
ordered list of child annotations and an unordered set of concurrent annotations. Each operation per loop iteration takes $O(1)$ with the exception of the Concurrent set operations. Those take $O(n)$, as each annotation has maximally $O(n)$ concurrent annotations to consider. As a result, the overall worst-case performance of this step is $O(n^2)$, with complexity trending towards linear with fewer concurrent tasks. Because humans tend to solve problems in sequential steps, expected average performance when applying this problem to single individuals solving problems is close to linear. Nonetheless, $O(n^2)$ is optimal for this problem, given the unlimited potential for n nodes concurrent to all other nodes, illustrated in 5.17. Given that each node requires assignment of each concurrent node and each node may have a list of n concurrent nodes, building concurrent lists requires $O(n^2)$ operations.

All ADD operations here append the node in question to the end of a list. Given the steps shown above, the algorithm generates the Children node lists in order according to each node's starting timestamp and only contains direct children — not descendants beyond that.

At this point, annotations are also assigned *Predecessor* and *Result* sets; these sets are initially empty and receive references to other annotations which subsequently map to Petri net edges. These sets describe what other annotations a given annotation leads to (*Result*) and what other annotations lead to it (*Predecessor*).

$$a \leftarrow \{ \textit{Predecessor}, \textit{Result} \}$$

Once hierarchical and concurrent relationships are calculated above, the Petri net generation algorithm applies the rules to detect sequential relationships as well. As show in Algorithm 5, the algorithm iterates through all parent annotations of interest — either all nodes without parents or single nodes manually selected by users and added to a list (if run in ‘single’ mode above rather than ‘all’) —



$$\begin{aligned} & a_x[Time] < a_{x+1}[Time] \\ D_{annotation} = \{ & a_1, a_2, a_3, \dots, a_x \} \mid a_{x_{end}}[Time] < a_{x+1_{end}}[Time] \\ & a_{x_{end}}[Time] > a_{x+1}[Time] \end{aligned}$$

Figure 5.17: For a set of n annotations, it is possible for each annotation to be concurrent to all other annotations.

and calculates predecessor and next annotations. The hierarchical and concurrent relationship detection step trivially places nodes without parents in $D'_{annotation}$ during iteration. During execution of sequential relationship detection algorithm, $D'_{annotation}$ receives child nodes as their parents calculate predecessors and results for them; this ensures child nodes have access to the appropriate predecessor node data from their parent nodes as they are iterated over and processed.

Subjects completing tasks have a point where they start and a point where they finish. In the annotations, these begin and end points are the bounds of annotations which do not have a parent — that is, top level annotations such as Task 1 in 5.10. In order to later provide a notion of directionality — important later in the visualization — default start and end nodes are placed at the beginning and end of these tasks. This is done in Algorithm 5 (1).

The algorithm iterates backward through its children. Each child is preceded by the non-concurrent children before it, found using a nested loop. This follows the pattern and rules illustrated in Figure 5.15. Any child nodes which do not lead to another child node necessarily result in the parent node; when a subject finishes these annotation tasks, they subsequently finish the parent to those annotations. These annotations that a subject completes last before completing the parent annotation necessarily consist of all the last nodes in the child list which are concurrent to each other — as an example, see Tasks 3 and 4 in Figure 5.14, which are the last concurrent annotations before the subject completes Task 1.

Children which have no other child predecessors (the set of children concurrent to and including the first starting child) are preceded by the parent's initial predecessor annotations — either the empty state if the parent itself has no parents, or the prior assigned annotations if the parent had it assigned through the ancestors' iterations; as an example, see Task 5 in Figure 5.16 which is assigned the predecessor of Task

Algorithm 5 Before lifting to Petri nets, predecessor and result nodes are calculated for each annotation. Default nodes are added which indicate users beginning or finishing a parentless annotation (broadly, a *task*). At this point, only *start* type nodes are in D and *end* nodes are examined only through links to the *start* nodes. Additionally, note that all parent nodes do not contain descendants beyond direct children in their [*Children*] lists.

```

//Iterate through all annotations, starting with top-level parentless nodes. Child
nodes are added to  $D'_{annotation}$  as their parents are processed and are iterated over
after their parents.

for Each  $n_{current} \in D'_{annotation}$  do

    ////(1) If the node does not have a predecessor or a result, then add the default
    one since this node is the starting or ending annotation.

    if  $n_{current}[Predecessor]$  EMPTY : then
         $n_{current}[Predecessor]$  ADD  $n_{empty}$ 
    end if

    if  $n_{current}[Result]$  EMPTY then
         $n_{current}[Result]$  ADD  $n_{complete}$ 
    end if

    ////(2) Iterate through the node's children.

     $i \leftarrow n_{current}[Children]$  length
    while  $i > 0$  do
        ////(3) For each  $child_i$  node, iterate through all earlier
         $child_{i-1}, child_{i-2}, child_0$  nodes to find its appropriate predecessor.

         $i' \leftarrow i - 1$ 
        while  $i' \geq 0$  do
            // A  $child_1$  node is preceded by other non-concurrent  $child_2$  nodes start-
            ing before it. First, check to see if the current  $child_2$  iteration ends before the
            max-starting predecessor  $child_2$  starts — in this case, the  $child_2$  is a predecessor
            of the predecessor of  $child_1$ .

            if  $n_{current}[Children][i'][Next][Time] <$ 
             $n_{current}[Children][i][Predecessor][0][Time]$  then
                //Because of the structure of the data, when the predecessor-of-the-
                predecessor is reached no other children can be direct predecessors to the  $child_1$ .
                 $i' \leftarrow 0$ 
            end if

```

```

if  $n_{current}[\text{Children}][i] \notin n_{current}[\text{Children}][i][\text{Concurrent}]$  then
     $n_{current}[\text{Children}][i][\text{Predecessor}] \text{ ADD } n_{current}[\text{Children}][i]$ 
     $n_{current}[\text{Children}][i][\text{Result}] \text{ ADD } n_{current}[\text{Children}][i]$ 
end if
 $i' --$ 
end while
// If no child predecessor is found, then assign it the parent's previously
assigned predecessor.
if  $n_{current}[\text{Children}][i][\text{Predecessor}] \text{ EMPTY}$  then
     $n_{current}[\text{Children}][i][\text{Predecessor}] \text{ ADD } n_{current}[\text{Predecessor}]$ 
end if
// A parent's predecessor is its children which do not have another child
as a result, though these are assigned at the end so the parent's default can be
passed to more children first.
if  $n_{current}[\text{Children}][i][\text{Result}] \text{ EMPTY}$  then
     $n_{current}[\text{FinalPredecessor}] \text{ ADD } n_{current}[\text{Children}][i]$ 
     $n_{current}[\text{Children}][i][\text{Result}] \text{ ADD } n_{current}$ 
end if
 $D'_{annotation} \text{ ADD } n_{current}[\text{Children}][i]$ 
 $i --$ 
end while
 $n_{current}[\text{Predecessor}] \leftarrow n_{current}[\text{FinalPredecessor}]$ 
end for

```

4, which is in turn the predecessor of Task 1, which is the empty set — Task 5 was assigned the predecessor from its most distant ancestor since none of its ancestors had another task as its predecessor like Task 3 is preceded by Task 2. As children are processed, they are appended to $D'_{annotation}$ to be processed later, so their parents predecessors are necessarily calculated before the children, which is necessary to ensure children select their predecessor from their ancestors accurately.

As above, this is a $O(n^2)$ step that trends to $O(n)$ as annotations trend towards sequential and hierarchical rather than concurrent: Each node is required to consider maximally n potential predecessors but only needs to find the first sequential predecessor; without concurrent predecessors, this is the first previous element.

Once the algorithm calculates the *Predecessor* and *Result* sets for each annotation, those annotations can calculate their respective cumulative states based on their predecessors using Algorithm 6. To do this, the algorithm starts at the first node and propagates forward its state. This component of the algorithm is computationally bounded by the number of edges between annotations, as each edge adds one element to the aggregated state. Here, there are $O(n)$ distinct edges: Each node in the graph adds exactly one edge coming from a source and one edge returning to the sink. Sequential and hierarchical relationships have direct flow between them, while concurrent relationships split flow; in either case, each annotation can only add one edge.

Once the algorithm completes aggregating the States, the algorithm splits annotations into States and Transitions nodes for a graph representation, with edges connecting them, as shown in Algorithm 7. Each node maps to the underlying annotation generating it. For each annotation, the algorithm creates a State and Transition node and connects them with an Edge from the Transition to the State. It also adds Edges from Predecessors, connecting the output state of each Predecessor

Algorithm 6 After determining predecessors, the algorithm calculates the state at each node.

```

 $D''_{annotations} \leftarrow EMPTY$ 
 $D''_{annotations} ADD n_{empty}$ 
for  $n_{predecessor} \in D''_{annotations}$  do
    for  $n_{result} \in n_{predecessor}[Result]$  do
        //The output state is the combination of this annotation's goal and the
        predecessor's output state, which includes all predecessors' goals.
         $n_{result}[State] \leftarrow n_{result}[Goal] \cup n_{predecessor}[State]$ 
        if  $n_{result}$  NOT YET CALCULATED then
             $D''_{annotations} ADD n_{result}$ 
        end if
    end for
end for

```

Algorithm 7 Once cumulative states are calculated, the algorithm splits nodes into Petri net states, transitions, and edges between them.

```

 $G_{states} \leftarrow EMPTY$ 
 $G_{states} ADD n_{empty}$ 
 $G_{transitions} \leftarrow EMPTY$ 
 $G_{edges} \leftarrow EMPTY$ 
for  $n_x \in D'_{annotation}$  do
     $G_{states} ADD s_x$ 
     $G_{transitions} ADD t_x$ 
     $G_{edges} ADD e_{t_x-s_x}$ 
    for  $n_y \in n_x[Predecessor]$  do
         $G_{edges} ADD e_{s_y-t_x}$ 
    end for
end for

```

to the input Transition of the initial annotation.

As in the previous step, this step of the algorithm is bound by the number of edges: $O(n)$. Once complete, the algorithm combines the resulting G_{states} and $G_{transitions}$ as G_{total} , the final graph nodes. Combining this with G_{edges} results in a graph representation of a Petri net built from a timestamped set of annotations in $O(n^2)$ time, with expected performance on real-world datasets close to linear. Beyond generating the Petri net itself, however, this algorithm also bounds as $O(n \log m)$ where m is $\text{length}(D_{total})$.³ Each node in the Petri net maps to a timestamp in the underlying data, which the algorithm leverages to bind each node to elements in other data sources. Binding nodes to the underlying data requires $O(\log m)$ per node to binary search for the appropriate indices to bind. Generating summary data requires $O(m)$ per node to iterate through each specified type underlying data type, as is the case in generating data such as active window titles or average process CPU load, resulting in $O(nm)$ total.

5.4.5 Petri Net Aggregation and Simplification

Petri nets generated from different time periods or sessions might contain identical Task and/or Goal data in annotations: This is to be expected particularly when comparing subjects doing the same tasks. The algorithm and rules presented generate one Petri net for each top-level parent annotation selected by the visualization user. Common states (in the aggregate) between these Petri nets may be equivalent, and as such can be combined into single nodes. Similarly, users might employ the same Tasks, which can likewise be combined by merging graph nodes. Much of this can be done automatically. However, doing so requires attention to annotation {Task,

³This is the length of the collected data, such as keyboard presses or screenshots, as shown in Figure 3.1. A selection of this data becomes associated with Petri net nodes.

Goal} naming schemes, in order to match correctly. Additionally, name collision may cause aggregation to occur when not desired, as discussed in 5.4.7. Allowing users to manually aggregate and deaggregate enables further data probing as well.

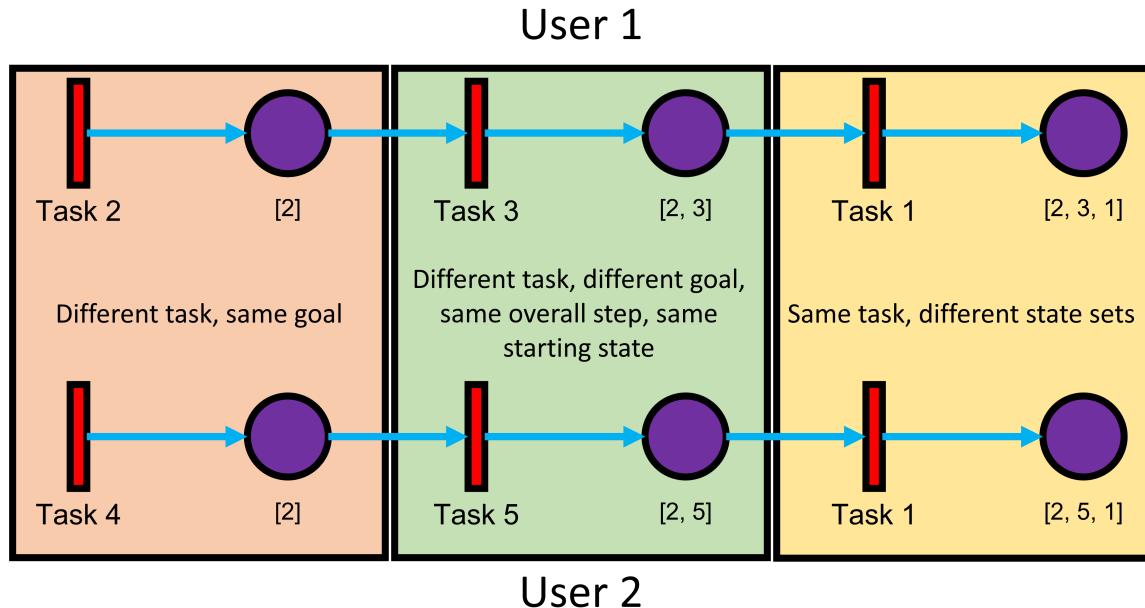


Figure 5.18: Commonality in different Petri nets enables users to compare those points to answer fundamental research questions.

Each aggregated node has differing underlying annotation data from which to mine information about the given task or state. This enables statistical analysis of the underlying data: Given two annotations with common tasks or states, as demonstrated in 5.18, it enables calculating distributions of time taken, keyboard/mouse input, common and contrasting process and window data, and analysis of any of the data in the Data Dictionary (See Appendix 9.6). All of this enables researchers to compare, contrast, and analyze resources needed to perform different tasks.

5.4.6 Petri Net Visualization

As mentioned, Petri nets consist of a set of transitions connected to a set of states, which in turn connects to a set of transitions. This trivially translates into a graph, with nodes consisting of transitions and states and edges where they connect. Petri nets, including the ones here, can be cyclic⁴, and they are directed. Our Petri nets all begin at a common start point (the empty state) but can end at both different states and common states.

Other work explores ways to visualize Petri nets as graphs [105]. Given the relatively small size of the Petri nets presented here and availability of ready-to-use libraries supporting the layout, a slightly modified force-directed layout was selected for the Petri net layout. [128]⁵ Transitions are represented as rectangles, while states are represented as circles. The singular start node is placed on the left side of the visualization, while each end node is placed on the right. End nodes are vertically distributed evenly along the right border. All other nodes are placed in the middle of the visualization with the force-directed algorithm applied to them to generate an easily readable layout, as shown in Figure 5.19.

After initial graph generation, a user can drag nodes around the visualization. Dragging freezes the dragged nodes in place but applies force to the other nodes to aid the user in configuring a layout that the prefer for the given graph. Glyph fill and stroke coloring can be utilized to encode different attributes to the visualization, and the amount of time taken and number of user inputs are shown in Figure 5.20. The annotation level is encoded into transition node size. Mouseover tooltips on the Petri net nodes show details⁶ for transitions and states; transitions are labeled with

⁴Cycles may appear in a variety of circumstance, such as when a particular task must be repeated multiple times before reaching its goal.

⁵The d3 library force-directed layout, specifically, was used for the visualization here. [310]

⁶Details include user input counts, time taken, and other statistics gathered from the underlying

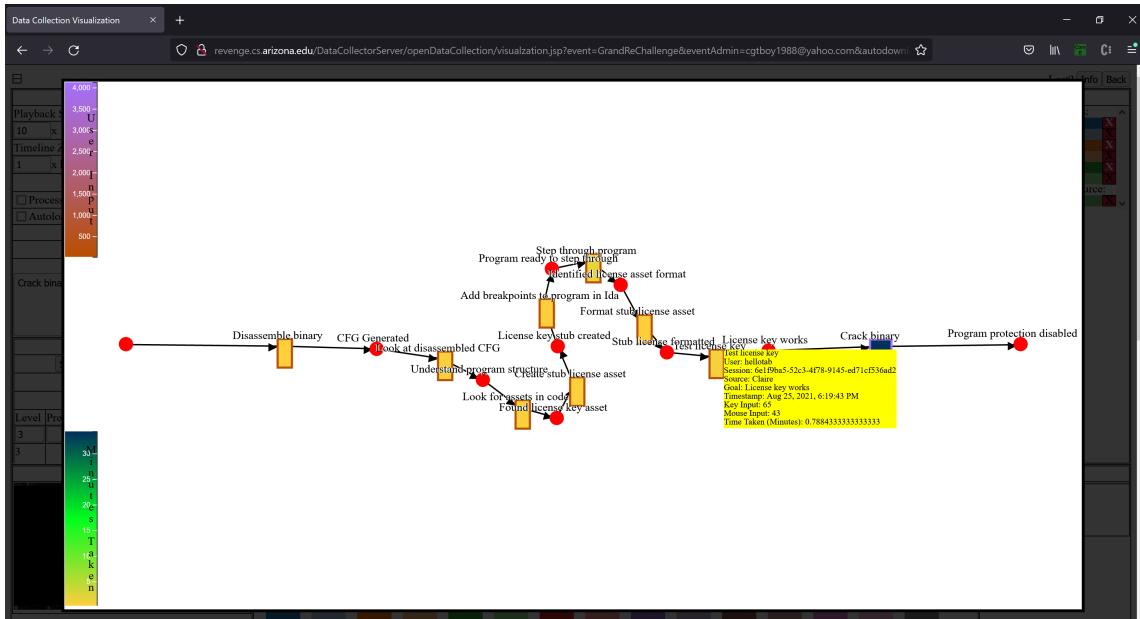


Figure 5.19: This screenshot of the Petri net visualization demonstrates the force directed layout for a single Petri net with a mouseover tooltip.

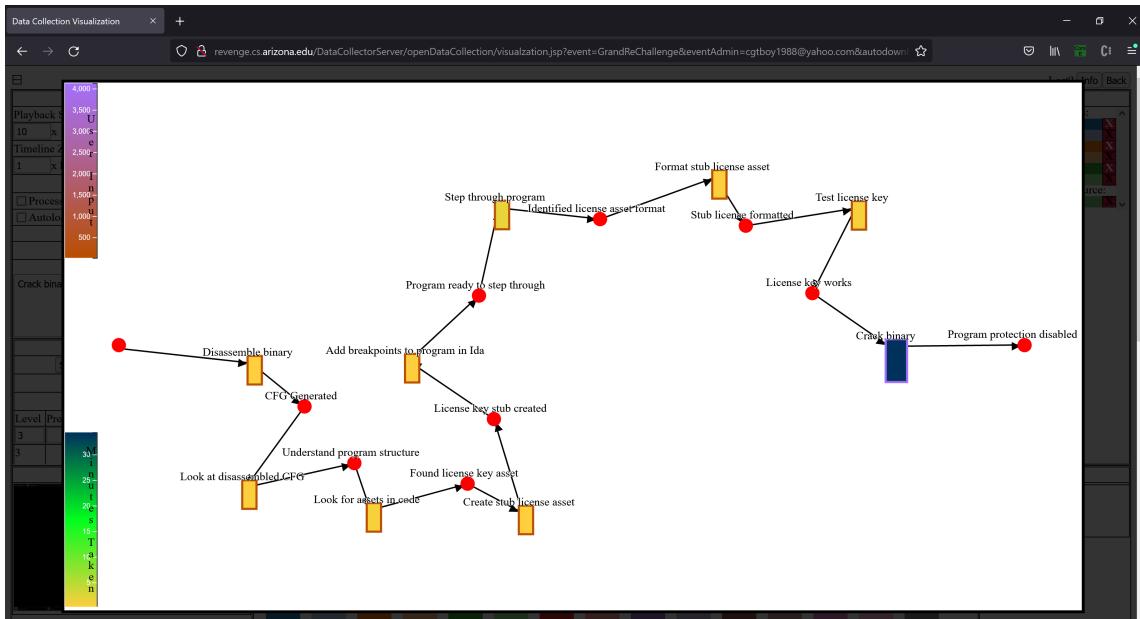


Figure 5.20: This screenshot of the Petri net visualization demonstrates the force directed layout for a single Petri net after editing the layout by dragging nodes.

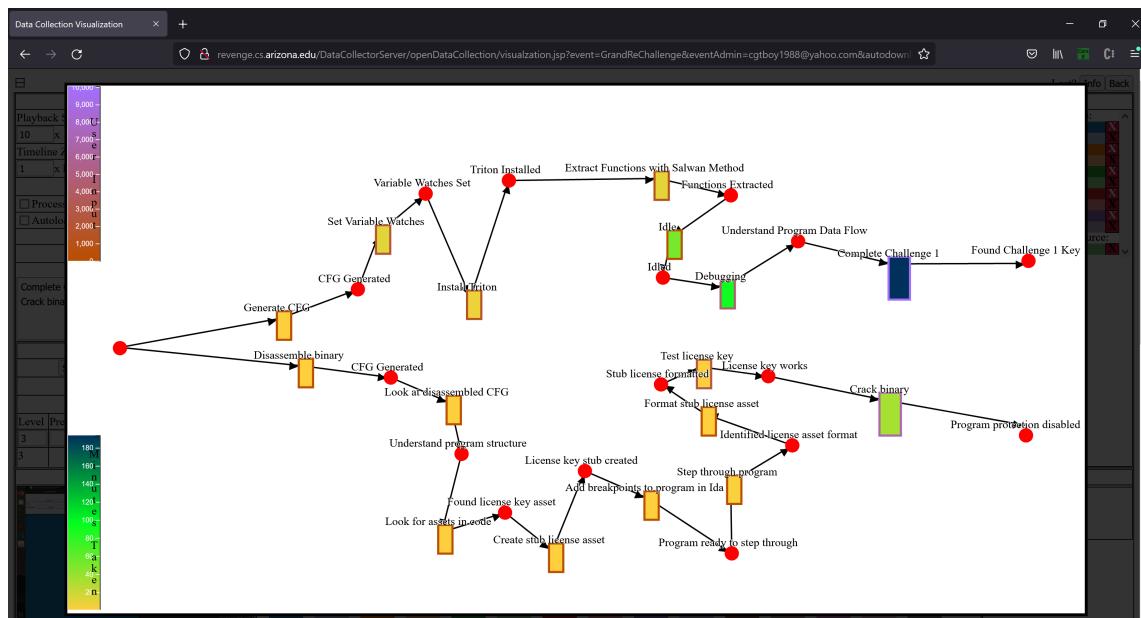


Figure 5.21: This screenshot of the Petri net visualization demonstrates the force directed layout for two Petri nets visualized at the same time with manual node position refinement.

the given task name, but states are only labeled with the the goal of the task (which is added to the overall state) rather than the entire state, as the latter can be quite lengthy and overflows the visualization. Left to future work is tuning of force-directed parameters and glyph/text sizing, as the current iteration of the visualization works well with the current data set size.

5.4.7 Nesting Considerations

Typically, tasks exist in the relationship patterns where parent tasks fully encompass child tasks and few concurrent tasks exist. People, in the context of using modern devices, do tasks relatively serially: They may have processes running in the background to complete tasks, but they only have one in-focus window that they are actively using and background processes are limited to the number a human can manage requires to complete their task. For reverse engineering in particular, the data analyzed thus far reflects a high degree of active window use with few processes requiring enough computation to run in the background. Given this, most tasks can be expressed as hierarchies. What these hierarchies look like can vary significantly, as there are many different levels of specificity and nesting that can be used when making annotations.

For instance, an annotation for a user generating a control flow graph and stepping through a program with the Ida debugger as a form of hybrid analysis might be labeled as in Figure 5.22(a). The corresponding hierarchy is shown in Figure 5.22)(b). Figures 5.23 and 5.24 show equivalent annotation hierarchies to Figure 5.22 with slightly simplified representations. These hierarchies can also be simplified by encoding information differently. As an example, entries like “Use Ida” can be encoded into individual annotations, as shown in Figure 5.25.

data.

Some annotations may also imply others, and their inclusion might be redundant. In the working example, one might surmise that “Hybrid analysis” is already implied by using both “Static analysis” and “Dynamic analysis”. Removing the former could further reduce the example hierarchy shown in Figure 5.25(b).

Further, it is possible to even include tags such as “Hybrid analysis” in order to preserve data fidelity. Tasks also have a "Note" field which allows annotators to enter high-level understandings of a given label. In this field, they can add natural language descriptions of the annotation, thus allowing editing and analysis for consistency and specificity.

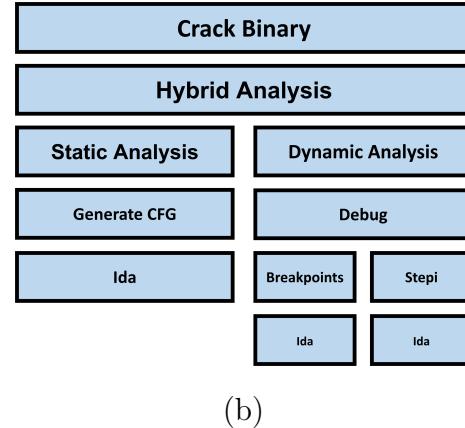
The tolerable level of ambiguity and resulting variance in annotations hinges on the application of the annotations. The research aims here require a high degree of fidelity and consistency in order to enable the types of comparisons and aggregations shown in 5.4.5. Moreover, the target audience for these annotations and models tends to have expertise in the subject matter — software reverse engineering. Thus, the question at hand is: **What level of model detail would reverse engineers find maximally beneficial?** That is, for a given annotation (a given step in a Petri net) what information would help reverse engineers understand how a subject reverse engineers a target? Based on the prior work describing reverse engineering methods (see Section 2.3) general strategies and prebuilt tool listings appear to be helpful. For novel methods where reverse engineers do more intricate and original actions, detailed descriptions of what they do (like published work describing individual novel methods) appear to be helpful in understanding these methods. It is not clear how close annotators get to this level of analysis from the HDI data analyzed, or how much effort it would take to get there.

Even with this normative analysis, finding the appropriate level for a given application is not easy without attempting to annotate data and build models. I discuss

the level of ambiguity and specificity required to analyze reverse engineering behavior in Section 6 based on the research team’s data annotation experience to date.

- Crack binary
 - Hybrid analysis
 - Static analysis
 - Generate CFG
 - Use Ida
 - Dynamic analysis
 - Debug
 - Set breakpoints
 - Use Ida
 - Step through program
 - Use Ida

(a)

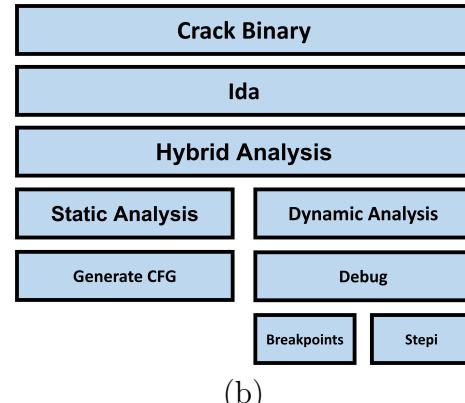


(b)

Figure 5.22: One possible iteration of an annotation hierarchy.

- Crack binary
 - Use Ida
 - Hybrid analysis
 - Static analysis
 - Generate CFG
 - Dynamic analysis
 - Debug
 - Set breakpoints
 - Step through program

(a)



(b)

Figure 5.23: An equivalent annotation hierarchy to 5.22 with a slightly simplified representation.

5.5 Summary

This chapter first describes the overarching goals of visualizing and analyzing Catalyst-gathered data in Section 5.1 before discussing performance constraints, given the

- Crack binary
 - Hybrid analysis
 - Use Ida
 - Static analysis
 - Generate CFG
 - Dynamic analysis
 - Debug
 - Set breakpoints
 - Step through program
- (a)

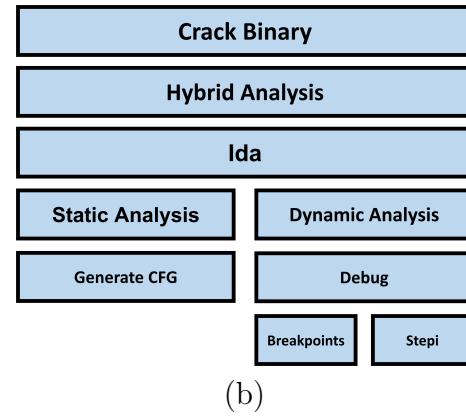


Figure 5.24: An equivalent annotation hierarchy to 5.22 and 5.23, demonstrating how the same activity can be represented in many ways.

- Crack binary
 - Hybrid analysis
 - Static analysis
 - Generate CFG (Ida)
 - Dynamic analysis
 - Debug
 - Set breakpoints (Ida)
 - Step through program (Ida)
- Crack binary
 - Static analysis
 - Generate CFG (Ida)
 - Dynamic analysis
 - Debug
 - Set breakpoints (Ida)
 - Step through program (Ida)

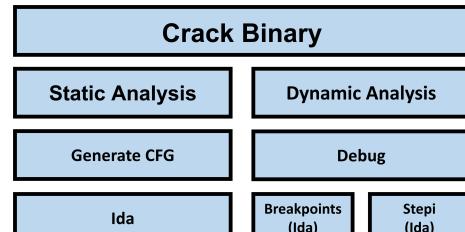
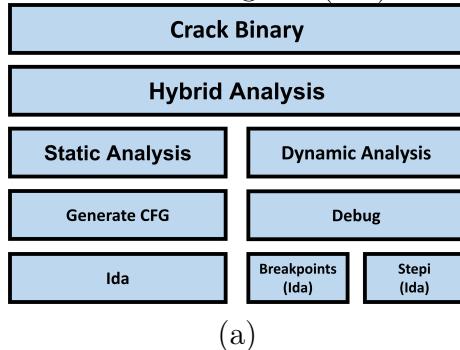


Figure 5.25: Removing “Hybrid Analysis” can simplify annotation hierarchies, as can integrating tools into other annotations rather than their own.

size and types of data in Section 5.2. The chapter then introduces the Catalyst visualization components, which enable analysts to annotate data in Section 5.3. Finally, the chapter specifies an algorithm which automatically lifts those annotations to human-understandable Petri net models of user behavior in Section 5.4.

Chapter 6

ANALYZING CHALLENGE DATA

The effort to analyze the collected data traces is ongoing, with lessons learned informing the process as it moves forward. In this chapter I will present what has been learned so far. In Section 6.1 we will discuss the current annotation progress and issues encountered, and present some Petri nets derived from the RevEngE CTF and Grand Re datasets. In Section 6.2 I show that the annotation process is fraught with ambiguity and that two analysts may approach the task in very different ways and reach different conclusions. Section 6.3 concludes.

6.1 Current Results

Thus far, only a few of the sessions from the *Grand Re* and *RevEngE CTF* have been annotated. Currently, two analysts have annotated 3 of the sessions (335 active minutes) from *Grand Re* winners, which corresponds to about 4.3% of the total active trace. Additionally, I have annotated the deobfuscation challenge from *RevEngE CTF*. So far, we have determined that annotation can be conducted roughly in real-time; i.e. for every minute of trace, we need one minute of annotation time. However, this varies significantly depending on the nature of the trace.

We have found that, for annotated sessions, the data collected by Catalyst is sufficient to gain an understanding of the methods employed by the subjects. We found the screenshot video view, as shown in Figure 6.2, to be the most effective tool in generating this understanding. Additionally, the analysts were able to translate those methods to annotations and ultimately generating Petri net models from them, as shown in Figure 6.3. However, due to the issues discussed in Section 6.2, this

initial work should be treated as a proof-of-concept in generating Petri nets: Until formal annotation guidelines are created enable analysts to generate consistent, well reviewed annotations, the annotation sets are limited in size and have not been extensively reviewed or evaluated. Moreover, the proposed guidelines aim to ensure the resulting annotations (and models built on them) conform to acceptable grounded theory coding standards, as described in Section 6.2.1.

6.1.1 *RevEngE CTF* Deobfuscation Challenge

The goal of the *RevEngE CTF* deobfuscation challenge was to deobfuscate and decompile an obfuscated binary to C source. The resulting trace consisted of a single session with over 200 hours of raw data, most of which was the device sleeping; of those 200 hours, roughly 19 hours contained activity. This particular session took roughly 6 hours to annotate, significantly less than the expected 19 hours.

The annotations and Petri net generated from them for the deobfuscation problem can be seen in Figure 6.1. The validity of the Petri net has not been formally validated to determine how well it models subjects' behavior — which would have required (at least) a post-event interview of the subject — or with regard to reverse engineering insights; this will be discussed further in Section 8.2. However, the subject made use of existing tools and methods to solve the challenge, and these were observable in the trace. Specifically, the subject used Ida with IDC scripting to step through the program (exploring different patterns of control flow) with scripts that extracted output behavior for input ranges. The subject used these ranges to reconstruct the program into non-obfuscated C source.

I encountered some issues with data fidelity in annotating; for instance, the data did not have enough precision to easily determine the exact timing of the subject's context switches between (quickly) writing script and running the debugger — though

the models may not benefit from such fine grained details regardless. Additionally, while I was able to observe the subject examining the radare2 reverse engineering tools, it was difficult to determine whether the subject successfully integrated them into the Ida interface or simply relied on the built-in Ida tooling.

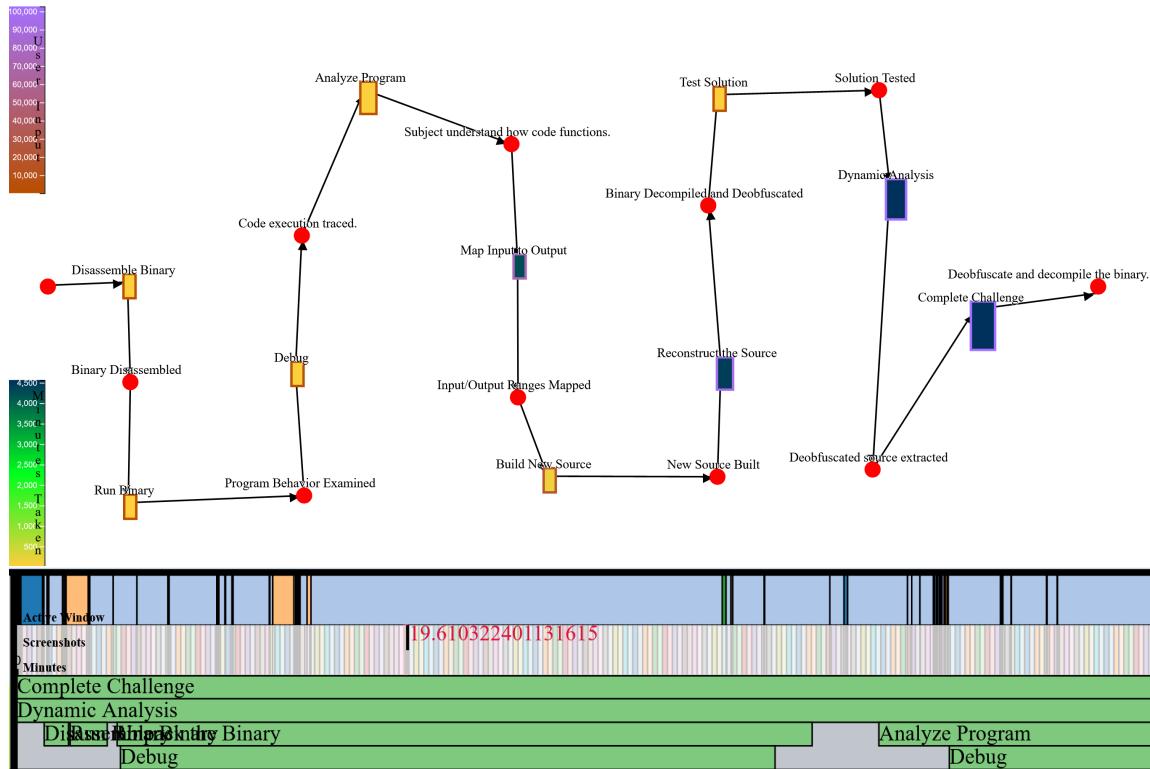


Figure 6.1: A Petri net generated from a successful *RevEngE CTF* deobfuscation challenge submission and a snippet of that submissions's annotations.

6.1.2 *Grand Re* Sessions

While the *RevEngE CTF* Petri net model is complete for the deobfuscation challenge, the *Grand Re* models built so far make sense but are incomplete. Only portions of subjects' methods have annotations so far which makes it impossible to draw solid conclusions from the trace. However, analysts have been able to discern activities

such as looking at control flow graphs in Binary Ninja, as seen in Figure 6.2, which fit into the incomplete Petri net models like the one seen in 6.3.

Grand Re subjects tended to work on several challenges concurrently. As a result, subjects generated many sessions over many days resulting in traces with intermingled challenged. Sorting through the traces and keeping track of annotations over all these sessions makes annotation more difficult and time consuming. Future events should ensure that there is clear demarcation between different challenges.

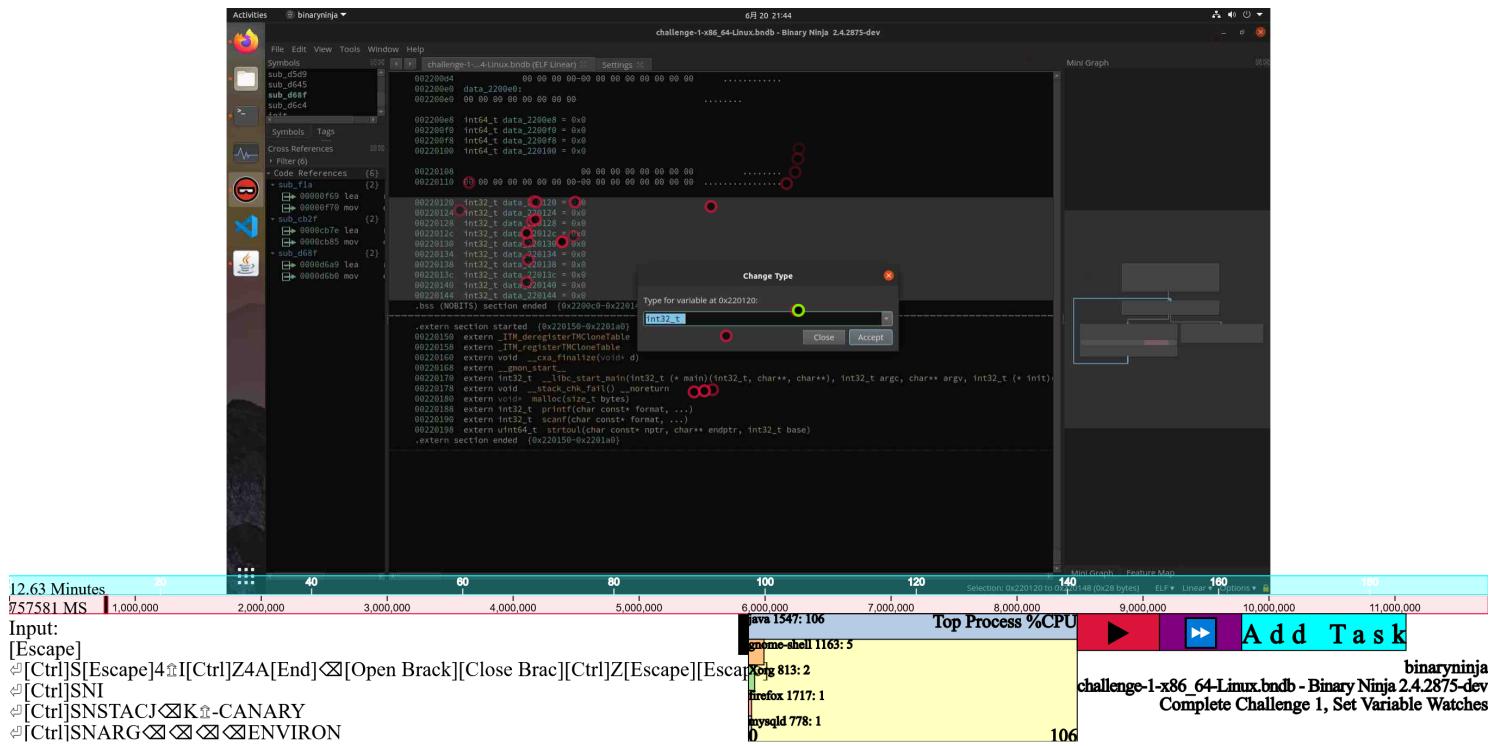


Figure 6.2: The screenshot video view of the top participant using Binary Ninja to look through Challenge 1's control flow graph.

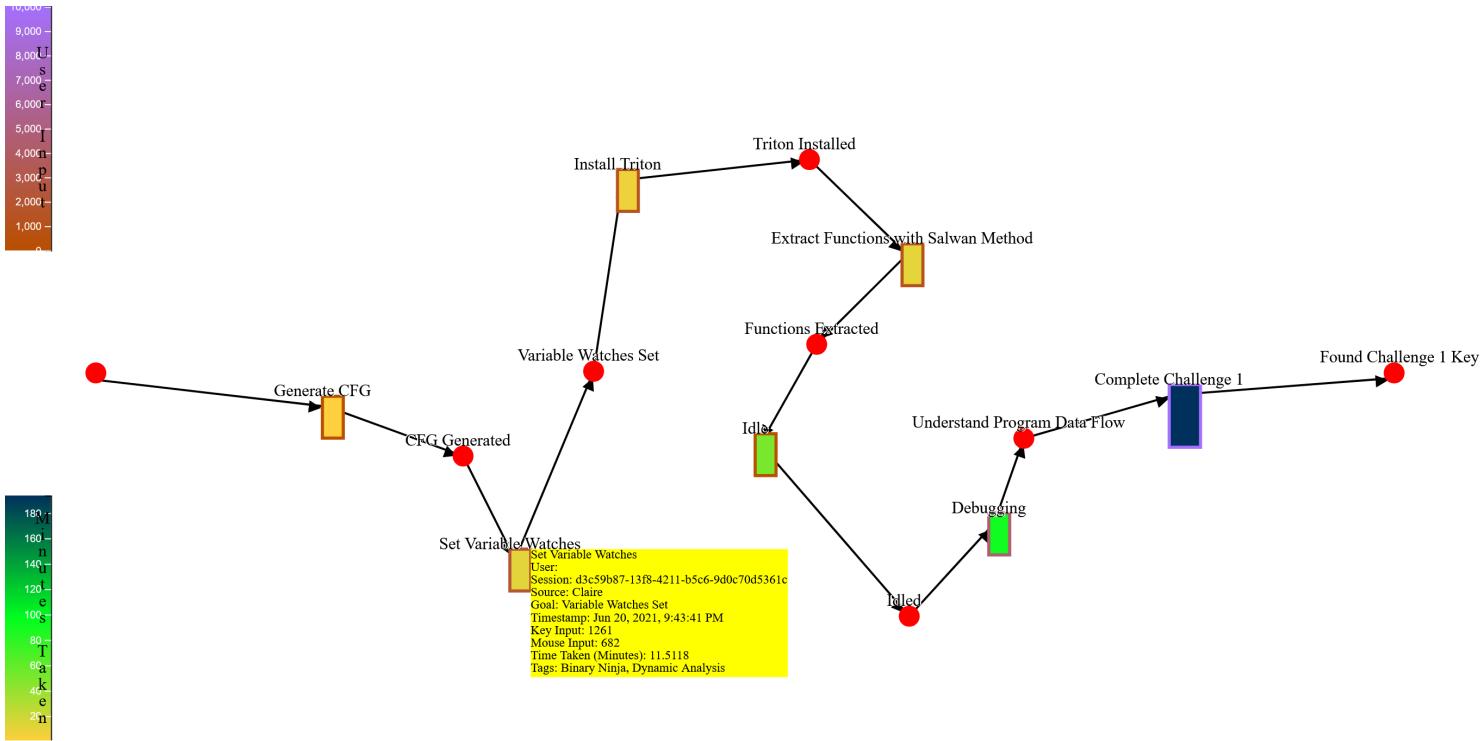


Figure 6.3: The resulting Petri net from the first session of the top participant working on Challenge 1.

6.2 Ambiguity in Annotation

Figure 6.4 shows a timeline of a few sessions of subject 3XNHKMGD of the *Grand Re*. Although 3XNHKMGD was eventually able to solve the challenge and win a prize, here we only show a subset of the work: 4 sessions completed over the course of 2 days. Note that the timeline is not to scale; rather, the 25 points represent the locations in the traces where the analysts determined that an event occurred for which an annotation needed to be added.

Two analysts, *A* (this author) and *B* (a PhD student from Ghent University) individually annotated the four sessions. *A*'s annotations are to the left in Figure 6.4, *B*'s to the right. Annotations are in red, goals in black.

It is easy to notice that *A* and *B*'s annotations differ widely. In particular, *B* does not add any serious annotation until timestep 17, while *A* starts earlier, for example by noting that the subject installed the Triton reverse engineering tool at timestep 4. When *A* and *B* share an annotation at a particular timestep, the annotation is either trivial (they both note that at timesteps 10 and 13 the subject turns to a particular challenge), or different (at timestep 12 where *A* simply says “debugging” and *B* says “Analysis w/ Binary Ninja and GDB”). Analyst *B* also tends to be more precise, noting the exact name of a file being modified (timestep 18).

Such widely diverging annotation styles is troubling. Consider, for example, if *A* and *B* were to annotate the traces of two different subjects solving the same challenge in order to determine multiple ways to approach the attack. Merging the two resulting Petri nets would yield no useful information since nodes denoting the same activity would likely diverge in labeling.

6.2.1 Grounded Theory Coding

Annotating datasets such as the one here in order to extract meaning is an example of grounded theory coding [137, 145] which attempts to build theory solely through the analysis of empirical data. The ambiguity encountered here is not uncommon for this type of problem, particularly when paired with highly detailed, high-fidelity data.¹ Salinger et al. [276] encountered similar issues with large annotation variance in initially annotating screen capture and subject-viewing video and audio from pair programming sessions. Annotating the data here does vary slightly from a traditional grounded theory approach inasmuch as the problem domain (reverse engineering strategies and methods) is somewhat informed from prior research and experience, but the problems Salinger et al. encountered in their annotation process (lack of focus, no standard granularity, descriptive vs assumptive annotations, etc.) closely parallel the ones discussed here. The lessons learned there can potentially aid the annotation efforts here.

Firstly, the grounded theory concepts (“phenomena, conceptualization, concepts, properties, categories, and relationships” which Salinger et al. in particular found useful) may be integrated into the visualization and annotation engine, either explicitly through new widgets and fields or implicitly by encoding them into existing fields by analysts. Secondly, we independently came to many conclusions similar to those found by Salinger et al. in developing strong guidelines for analysts to follow. Thirdly, we also believe that some basic automation methods, many of which have already been explored in prior work as detailed later in Section 8.1, can further aid annotation efforts in increasing consistency and decreasing annotation time.

¹By contrast, asking students to “think aloud” while tracing through code does not generate a similar extent of ambiguity issues in grounded theory coding-based annotation [125].

6.2.2 Annotation Guidelines

Salinger et al. propose using a predefined "perspective" which informs three research inquiries:

1. In which respects do you expect the data to provide insight?
2. What kinds of phenomena do the researchers allow themselves to identify in the data?
3. What type of result do you want the analysis to bring forth?

They also propose naming conventions and syntax, as well as pair coding in order to achieve better consistency.

Given our independent observations which are consistent Salinger et al.'s initial annotation attempt findings in many respects, we believe it is essential to create more robust annotation guidelines for the analysts to follow which include the listed recommendations. Ideally, such guidelines would be integrated into the annotation tool, gently guiding the analyst to follow accepted conventions while annotating traces. We have started to do so, by providing the analyst with a list of *standard labels* (pane (A) in Figure 5.5) to choose from. The current set of labels were derived from the experiences gained by the ASPIRE project [79], but we expect the list will need to be revised over time.

We also believe there will be a need for post-annotation conference between analysts if pair-coding is not possible,² where they review each others' annotations to ensure consistency, adherence to naming conventions, and to discuss if new standard labels need to be added to the guidelines.

²Unlike Salinger et al.'s work, the analysts are located in geographically distant locations, though some pair-coding activity may be possible with real-time screencasting technologies.

6.2.3 Automating Annotations

We further believe more annotation automation to be necessary. For example, if traces can be automatically segmented (broken into pieces at times when the subject changes major activity) then we can force the analysts to put annotations at the beginning and end of such segments. Similarly, if certain types of common events can be programmatically identified (such as “click STEP in Ghidra”) then the corresponding annotations can be automatically inserted. Together, such automation can provide a scaffolding onto which the analysts can hang more detailed annotations; this will hopefully guide multiple analysts to arrive at similar sequences of annotations. We will discuss such automation in future work (Section 8.1).

6.3 Summary

This chapter details some of the issues the research team encountered in annotating The Grand Re dataset in Section 6.2 and displays some of the initial annotations and Petri net results in Section 6.1. Future work ought to focus on some of the issues found, particularly considering ambiguity in annotating Catalyst data. Efforts to annotate RevEngE CTF and Grand Re datasets continue and evaluation of current annotations and Petri net models is needed. Nonetheless, these initial results and experience reveal strategies that use existing tools and methods which participants used in solving reverse engineering challenges.

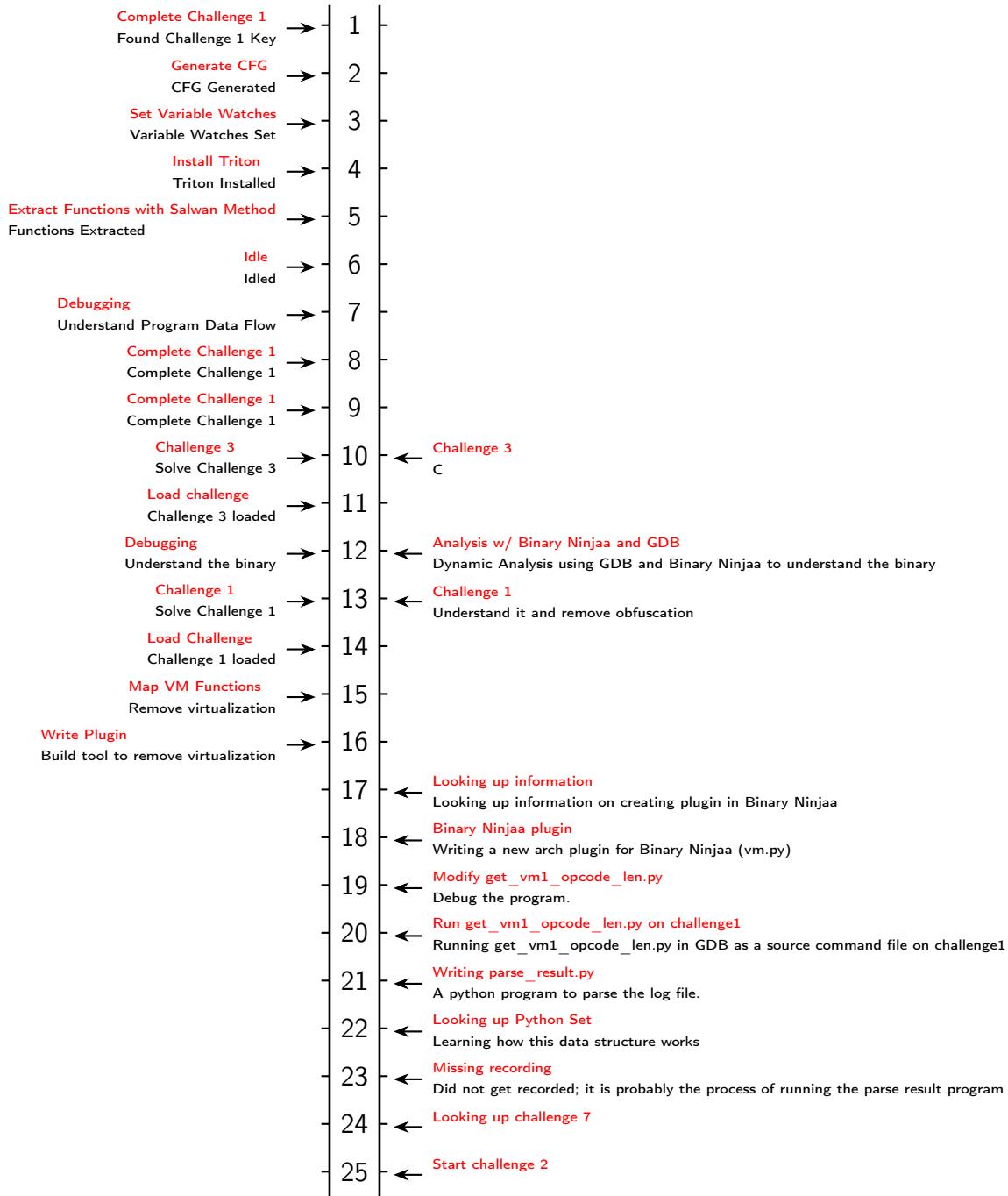


Figure 6.4: A timeline of the annotations provided by two analysts (left and right of the timeline) of 4 sessions from the *Grand Re*. Annotations for events that happened at approximately the same time are shown as happening simultaneously.

Chapter 7

PERFORMANCE ANALYSIS

Research Question 1 asks: *Is it possible to build a system to remotely and unobtrusively collect high fidelity HDI data from experts in their own heterogeneous computing environments as they reverse engineer software?* This points to a potential issue with a system like Catalyst: can the endpoint monitor collect data without being intrusive, given that it will run on subjects' own systems, with potentially very different performance characteristics?

Subjects running Catalyst for RevEngE and The Grand Re generated traces and did not complain about endpoint monitor performance. Nor did questionnaires to subjects who did not complete challenges generate any responses that indicated performance issues. However, students from The Ghent University study (Section 4.9) indicated that the endpoint monitor could sometimes introduce user interface lag and choppiness and that their virtual machines sometimes became unusable while running Catalyst. Later review of the data demonstrated data fidelity issues related to poor performance in some environments.

In this chapter we will discuss the performance aspects of Catalyst. Section 7.1 discusses the necessary monitoring facilities, Section 7.2 the experimental setup, and Section 7.3 results. Section 7.5 summarizes.

7.1 Monitoring Facilities

Catalyst has been given a profiling API to allow subjects' performance issues to be investigated. Latency issues for different components, such as screenshot generation and database writeouts, are particularly problematic and need to be monitored.

I also added thread-level granularity to the Catalyst process monitoring component. Process monitor datapoints record attributes such as CPU and memory usage on an interval in Catalyst. This thread-level granularity, however, introduced a significant storage overhead unsustainable on the current hardware. As such, the process monitor required a different approach which reduced disk usage. The new approach involved a sparse datapoint recording optimization. However, many of these datapoints are redundant: If no attributes change (CPU and memory use stay the same, for instance) then the datapoint does not add any new information; rather, it repeats the prior datapoint. As such, these redundant datapoints need not be recorded. At the same time, simply not recording identical datapoints at all removes important information — specifically, this level of sparsity removes process death information from the dataset. In the prior approach, process death is apparent when a process no longer appears after a given interval. But not recording processes on that interval (as is the case when a process repeats under the sparse recording method) removes the last datapoint and knowledge of when a process died.

Catalyst resolves this problem by maintaining a last-known-value for each process. When a process in the last-known-value set does not appear in the latest interval, Catalyst knows that the process died and records a datapoint indicating this. When the latest interval finds a process not contained in the last-known-value set, Catalyst knows that the process is newly born and records a datapoint. Finally, Catalyst knows processes appearing on both sets simply continue and only require a datapoint if they have significant changes. These three categories can be seen in 7.1. Implementing this optimization reduces the amount of storage required for process monitoring generally and enables thread-granularity on current deployments.

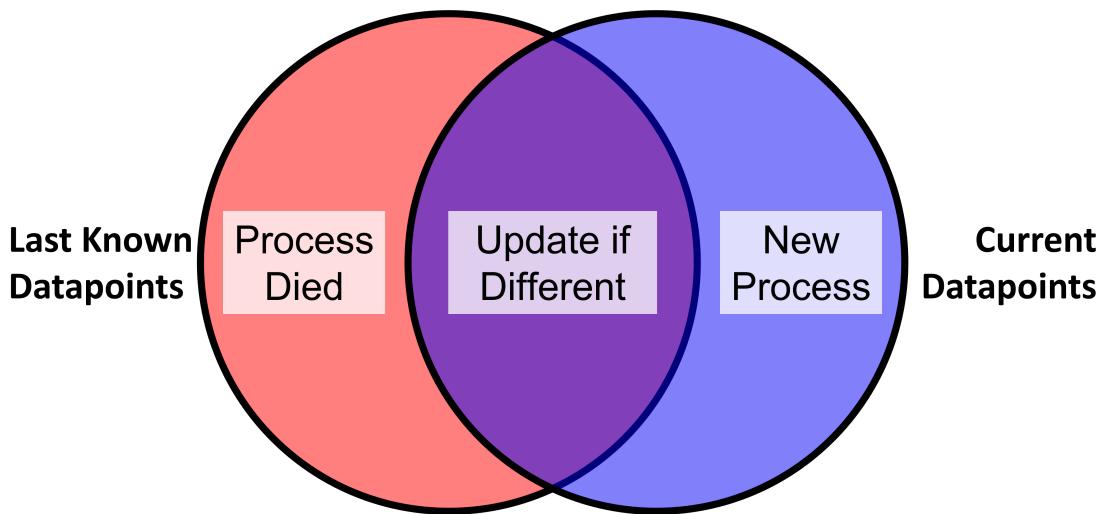


Figure 7.1: The process monitor now uses a sparse approach that only records processes as they are born, die, or have significant updates. This set intersection calculation completes in $O(n)$; the intersection calculation operates on the set of current datapoints and the set of datapoints recorded in the prior process monitoring sample. All datapoints in the blue and red categories are stored, while the purple category datapoints are stored if significantly different (in CPU and/or memory use) than the previously stored datapoint from the same process.

7.2 Experimental Setup

Armed with performance metrics and thread level monitoring, I examined performance on two devices with two different operating systems — specifically, Ubuntu 21 (running on Xorg for compatibility) on a VMWare virtual machine with 4 GB of memory and 3 CPUs allocated, and a Raspberry PI 4 (RPI4). The VM used a 4k (ultra high definition, UHD) display, with roughly 2600x1600 pixels allocated, though the exact pixel counts varied slightly between runs. The RPI4 ran on a standard 1080p (high definition, HD) display. I ran each environment three times, with different process and screenshot collection intervals¹ in order to determine roughly how much performance can be attained and at what system resource cost. Those three settings are as follows:

- Process: 20000ms; Screenshot: 0ms
- Process: 2000ms; Screenshot: 1000ms
- Process: 5000ms; Screenshot: 2000ms

7.3 Performance Results

In this section I will discuss some interesting findings from the experiments.

7.3.1 Endpoint Monitor CPU and Memory Usage

Figure 7.2 illustrates how CPU usage changes over time. It can be seen that CPU use peaks early, as is expected for a Java application where the virtual machine launch generates early overhead. Note that Kali RPI CPU use significantly increases with process monitoring interval decreases, even as the screenshot interval increases.

This is not the case with the Ubuntu VM, as shown in Table 7.1.

¹Collection intervals determine how long a thread sleeps in between polling data.

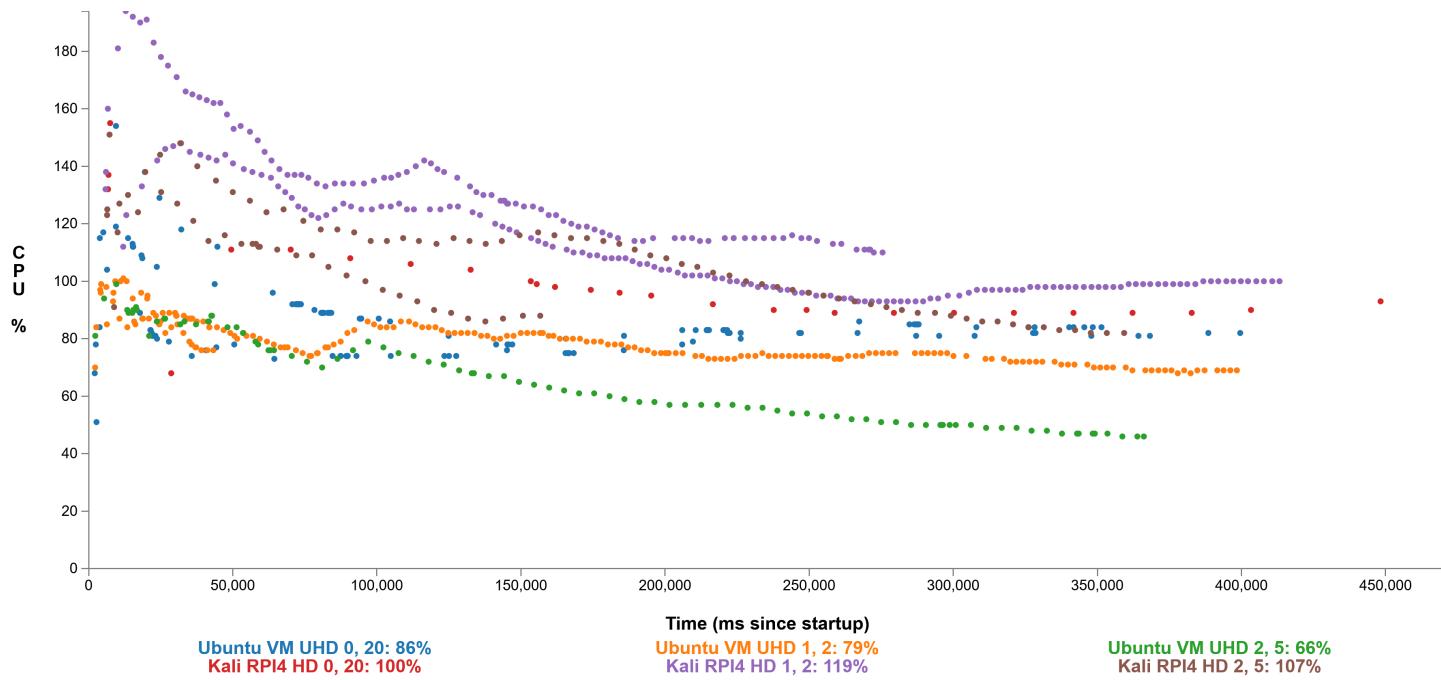


Figure 7.2: CPU usage over time, since startup of the endpoint monitor. Environments are listed on the bottom legend; the two numbers are the interval (in seconds) to sample screenshots and process data, respectively.

Figure 7.3 shows x86 using more memory than RPI4, likely due to capturing a higher display resolution. Except for a few memory use spikes (which warrant further analysis) memory use appears acceptable generally.

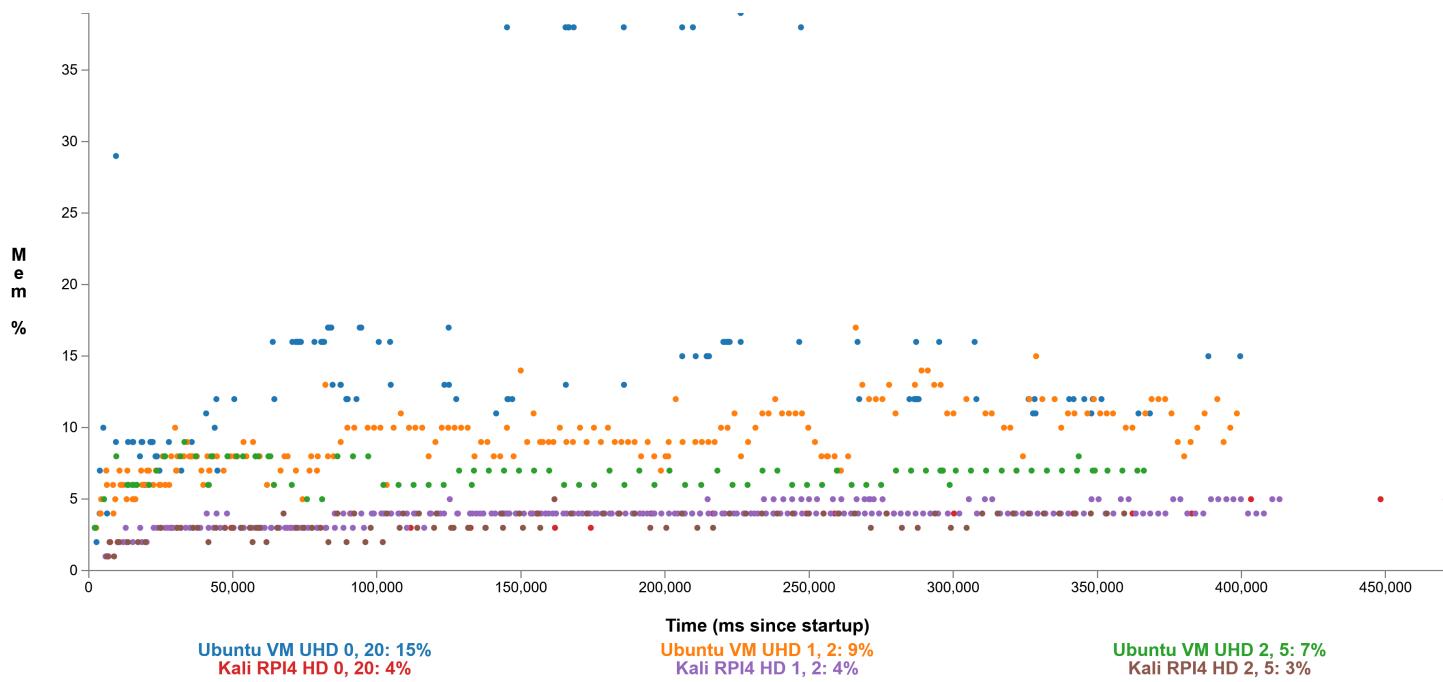


Figure 7.3: Memory usage over time, since startup of the endpoint monitor. Memory is measured on the operating system as a percentage of system memory.

7.3.2 Endpoint Monitor Thread Usage

Figure 7.4 indicates that in the VM setup, screenshot generation tends to use the largest amount of CPU, with upwards of 50% use on the high resolution display when running with the minimal interval of 0 ms. Process monitoring also requires a large amount of CPU resources, using about 10% of the CPU while screenshot generation on the same 2000 ms interval uses 20%.

Unlike the Ubuntu virtual machine data, the Kali RPI dataset in Figures 7.5 indicates more impact from process monitoring, with screenshot and process monitoring threads having roughly the same 20% CPU use at the 2-second polling interval.

Figure 7.6 shows a direct comparison between RPI data and VM data for the main data collection threads (Screenshot generation, process monitoring, local data writing, data export, and IO reading). This indicates that screenshot generation utilizes the largest amount of CPU in both cases, while process monitoring uses significantly more (as a percentage of the total process use) CPU for the RPI.

In summary, the RPI generates higher Java CPU overhead and tends to have more trouble scaling to lower process sampling rates, while the VM setup tends to have more screenshot-based CPU scaling issues.

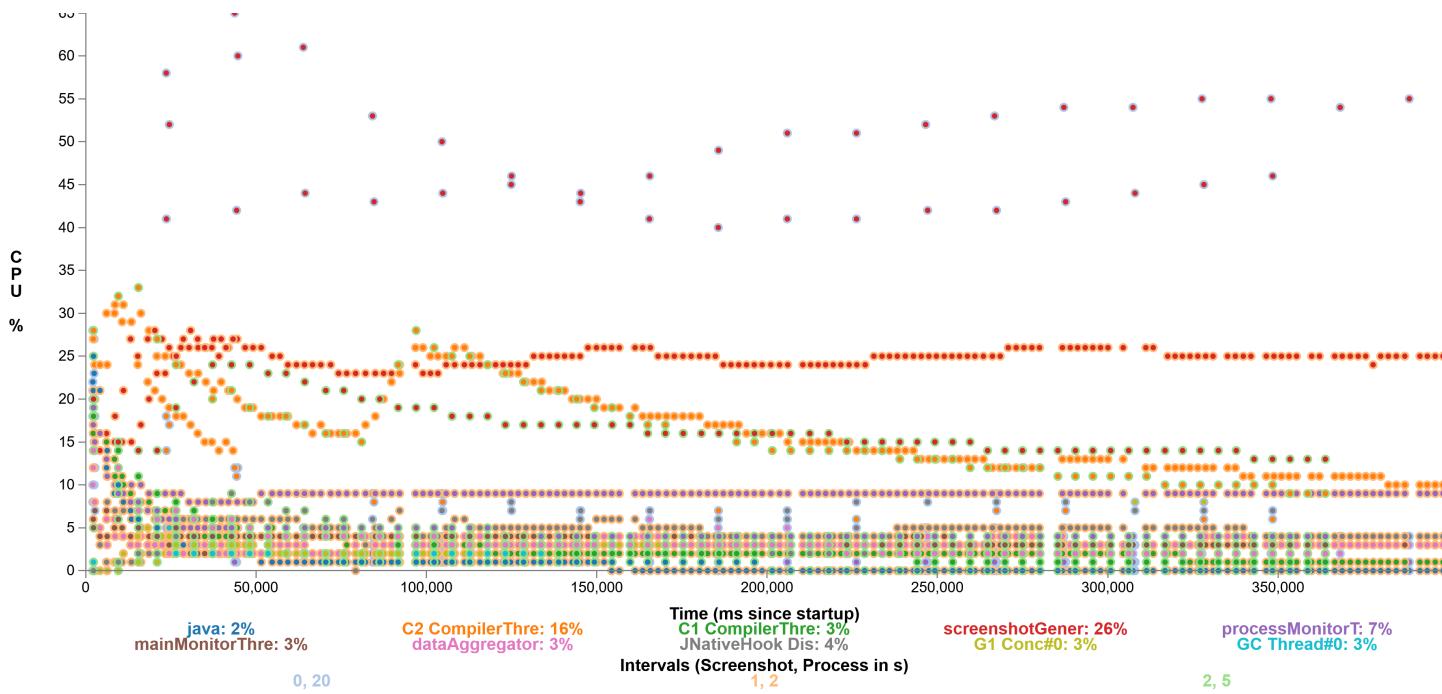


Figure 7.4: Percent CPU usage of individual threads when operating in the testing VM setup. Different intervals (color coded by the circles' stroke) demonstrate the performance impact of different settings.

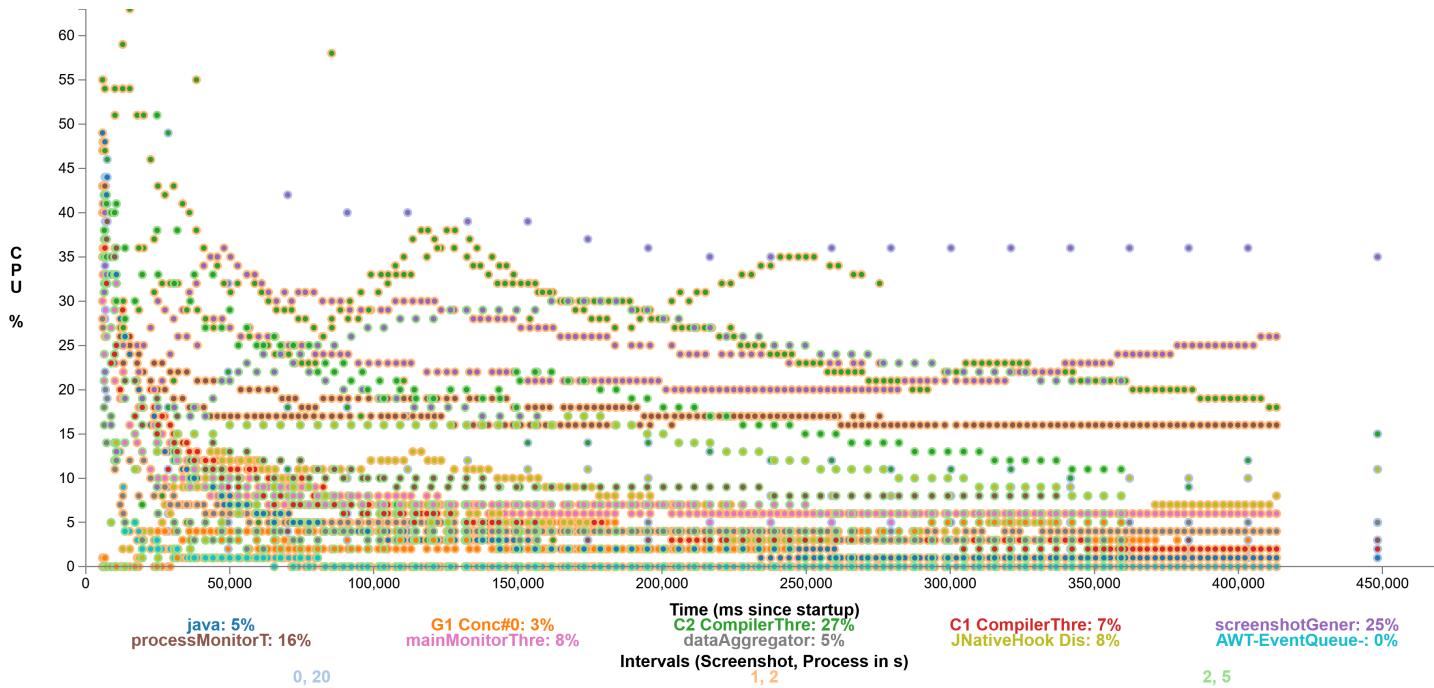


Figure 7.5: Percent CPU usage of individual threads when operating in the testing Kali RPI4 setup.

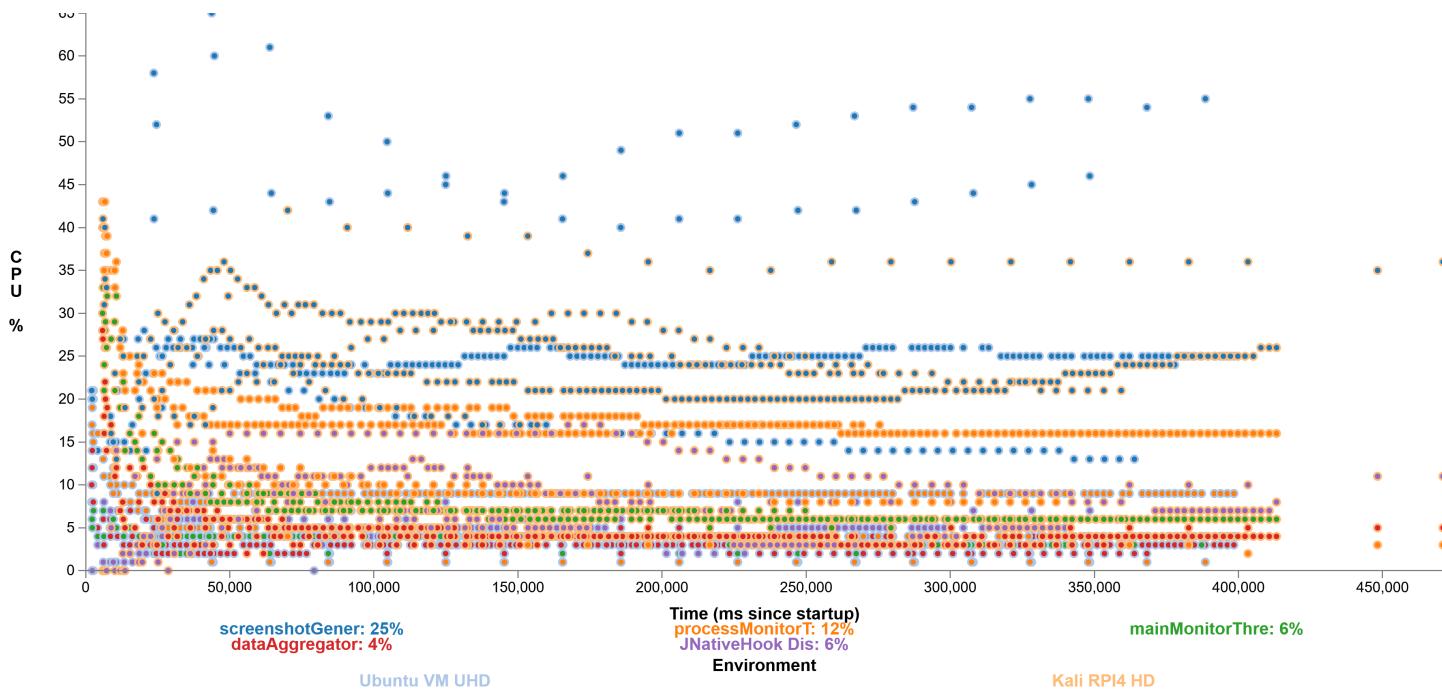


Figure 7.6: RPI data and VM data for the main data collection threads (screenshot generation, process monitoring, local data writing, data export, and IO reading).

7.3.3 Endpoint Monitor Page Faults

Page faults occur when the Endpoint Monitor loses memory locality and needs to remap or load data from slower storage sources [110]. Too many faults can lead to inconsistent performance as the program hangs awaiting memory access. For the VM setup, Figure 7.7 shows a significant number of minor faults with the screenshot generation and process monitoring, increasing with decreased sampling intervals, as expected. Major faults in the virtual machine setup (Figure 7.8) are generally the result of screenshot generation or built-in JVM overhead.

The RPI follows the pattern set by the VM with regard to minor faults (Figure 7.9). The RPI, however, experiences few major faults (Figure 7.10) relative to the VM setup, which is expected given the overhead and memory management issues virtualization can introduce.

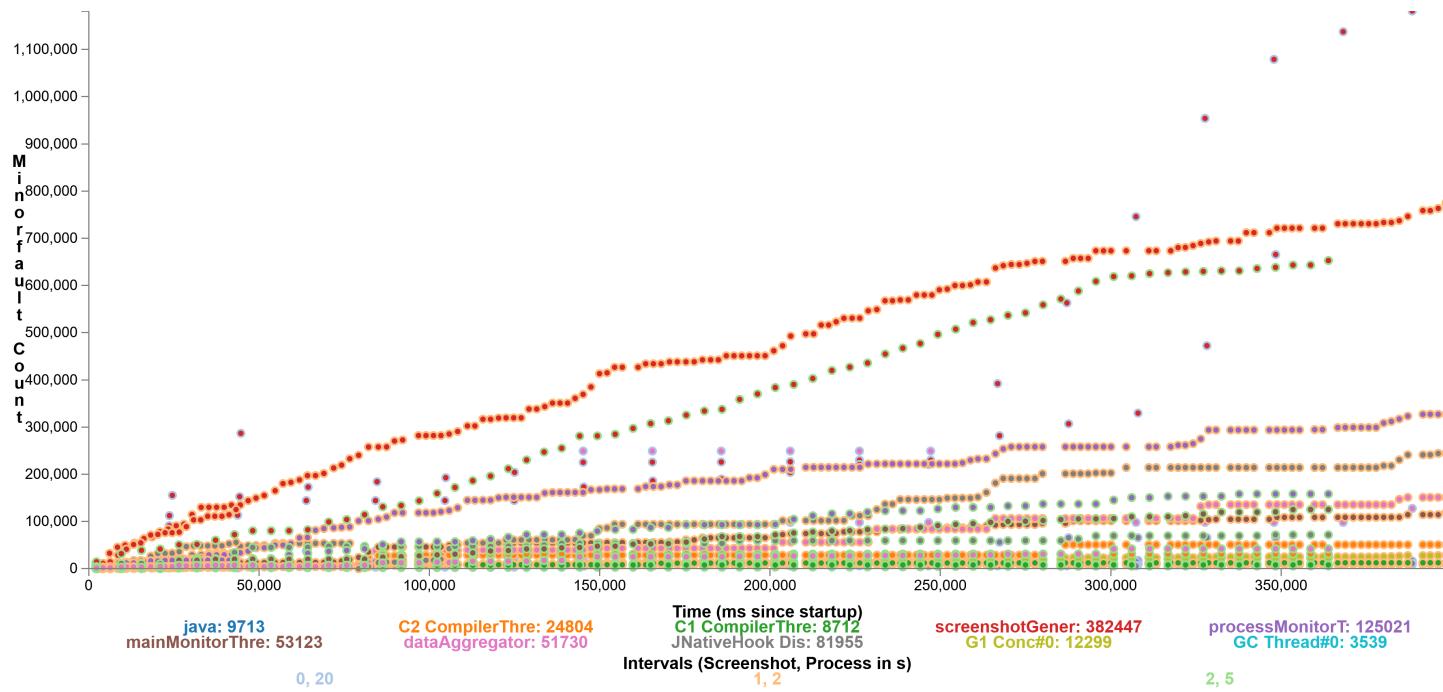


Figure 7.7: The minor fault count for the virtual machine setup.

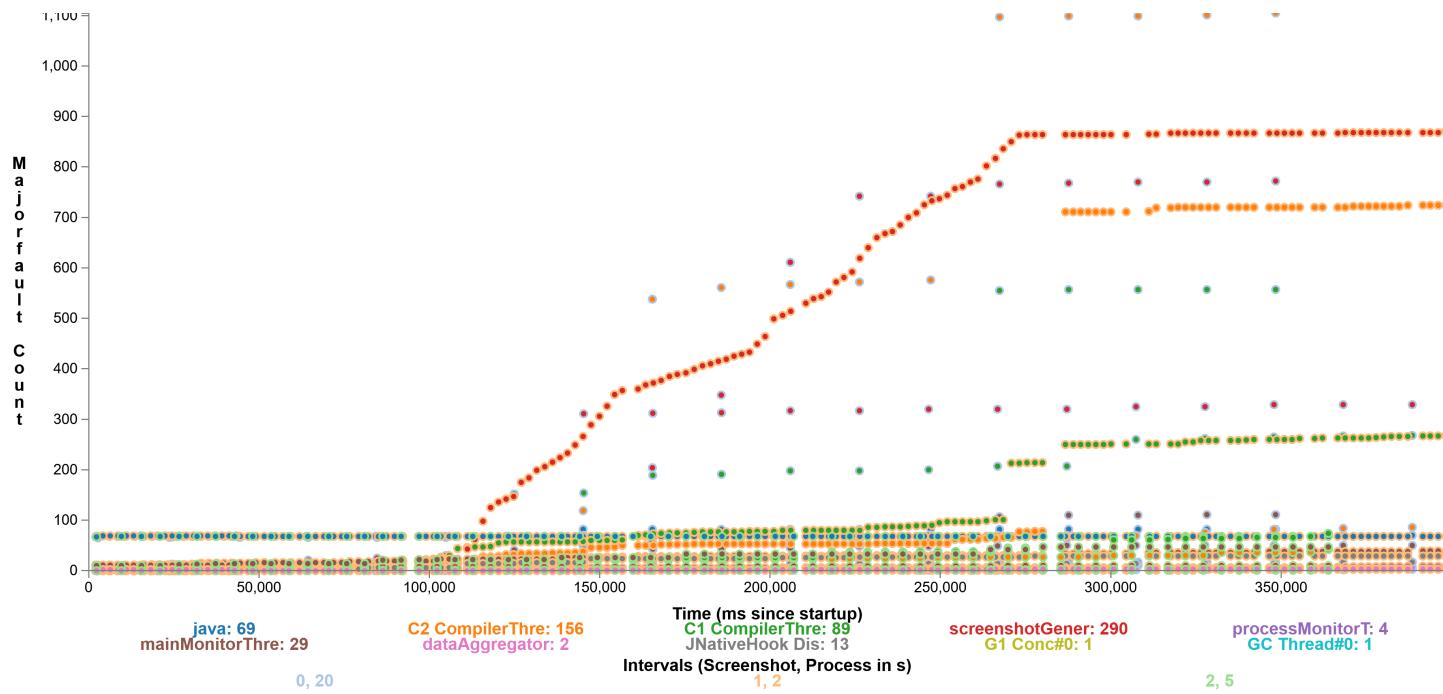


Figure 7.8: Major faults in the virtual machine setup.

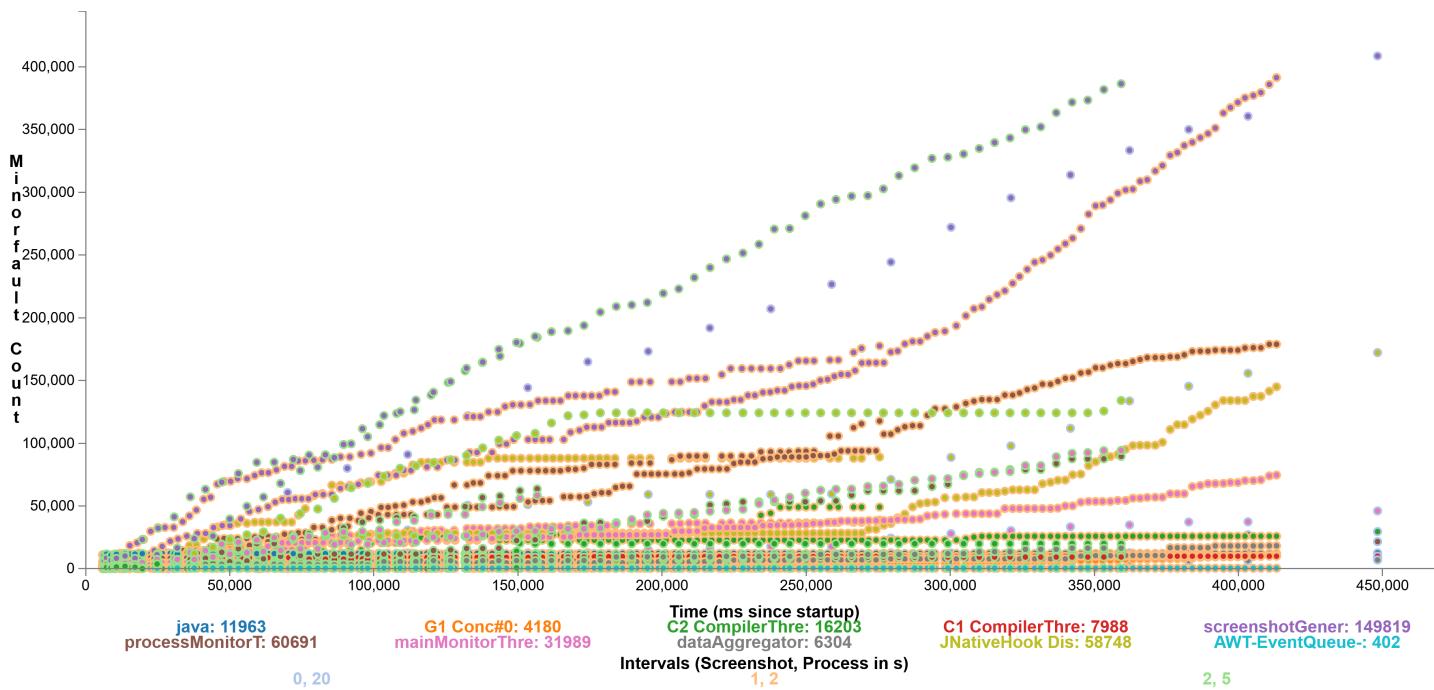


Figure 7.9: Minor fault for the RPI setup.

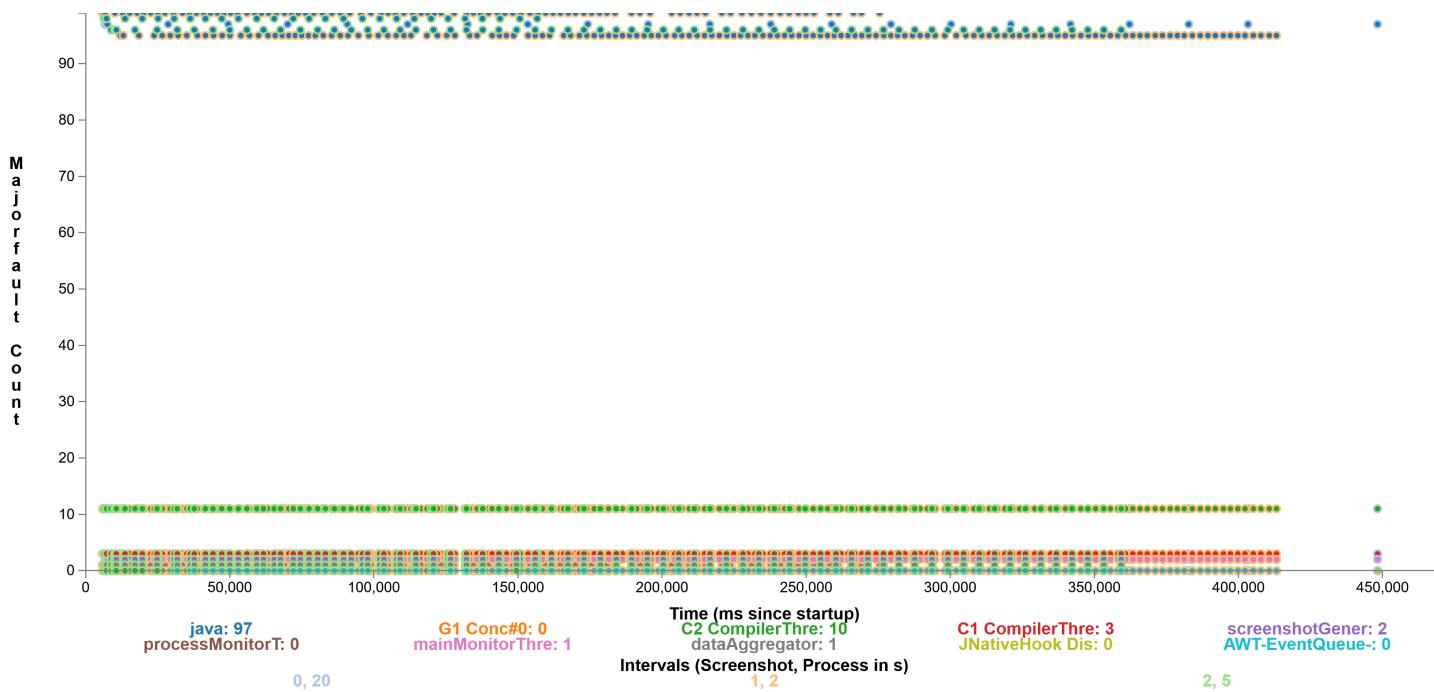


Figure 7.10: Major faults for the RPI setup.

7.3.4 Achievable Write Rates

Figures 7.11 and 7.12 demonstrate that screenshots tend to have the lowest sample performance, both in generation and in storage. RPI4 tends to have lower persistence performance, but otherwise the storage patterns correlate well to the complexity of data entry sizes. Most interestingly, Figure 7.12 shows that one screenshot per second is achievable both in the VM and RPI settings.

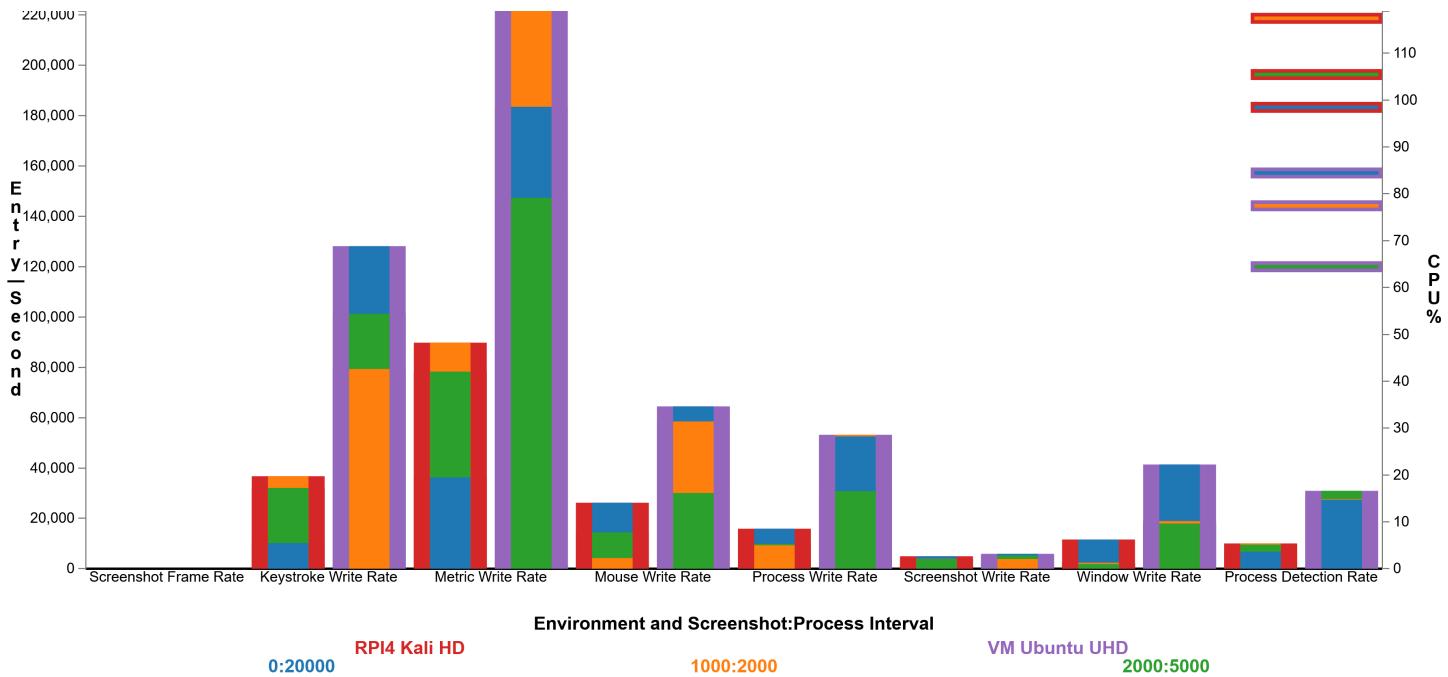


Figure 7.11: Amount of execution time taken by different endpoint monitor components under different environments and settings, alongside average CPU use. Except for the screenshot frame rate, these are all directly measured within the endpoint monitor. Screenshot recording and window detection proved to be too quick to measure without more precise (and computationally expensive) timers, as they measure only a single data point per metric. However, the average screenshot time here is calculated based on entire sessions data rather than internal metrics.

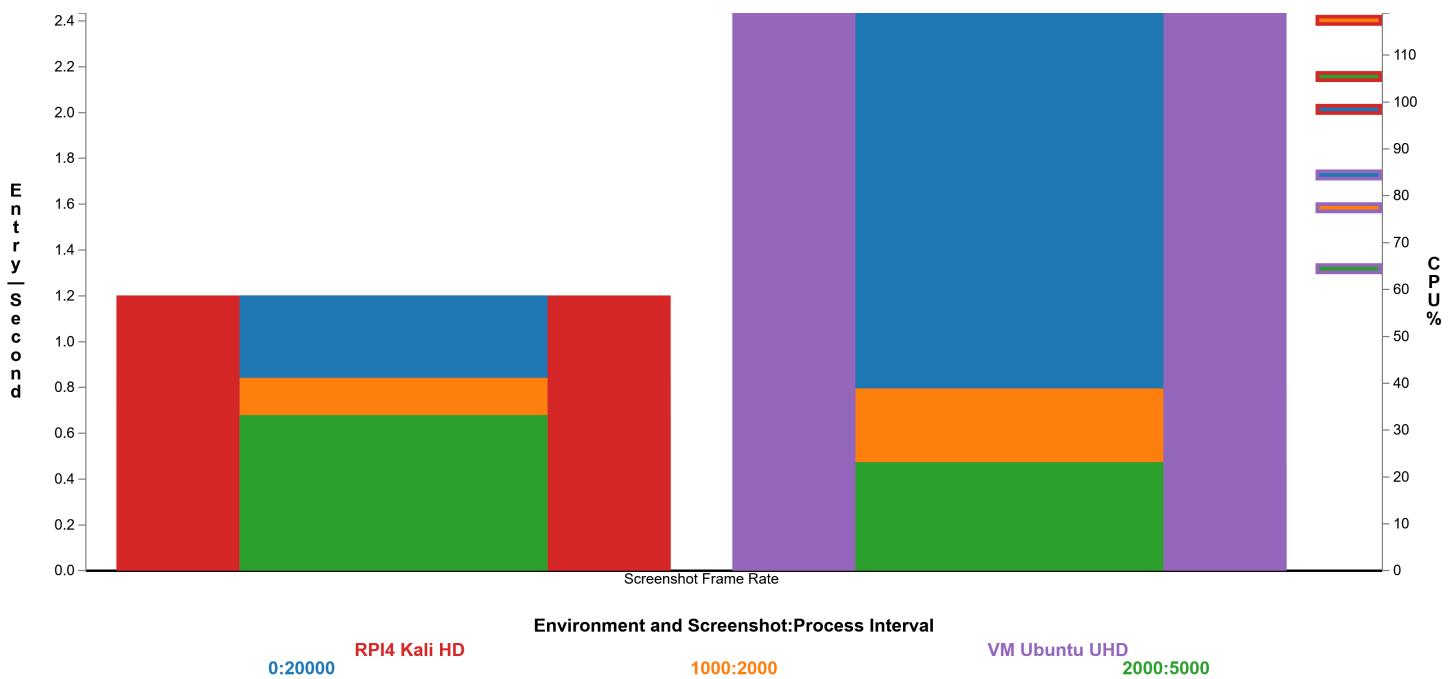


Figure 7.12: A zoomed in view of Figure 7.11 focusing on the screenshot frames per second metric.

7.4 Persistent Storage Performance

Figures 7.13 and 7.14 reflect persistent storage performance when scaling; in all cases, writeouts kept up with data generation and the graphs demonstrate that the endpoint monitors scales adequately in test cases with additional data. Figure 7.14 shows that the RPI running Ubuntu at HD tends to scale with screenshot data, while the VM with UHD scales more with process data.

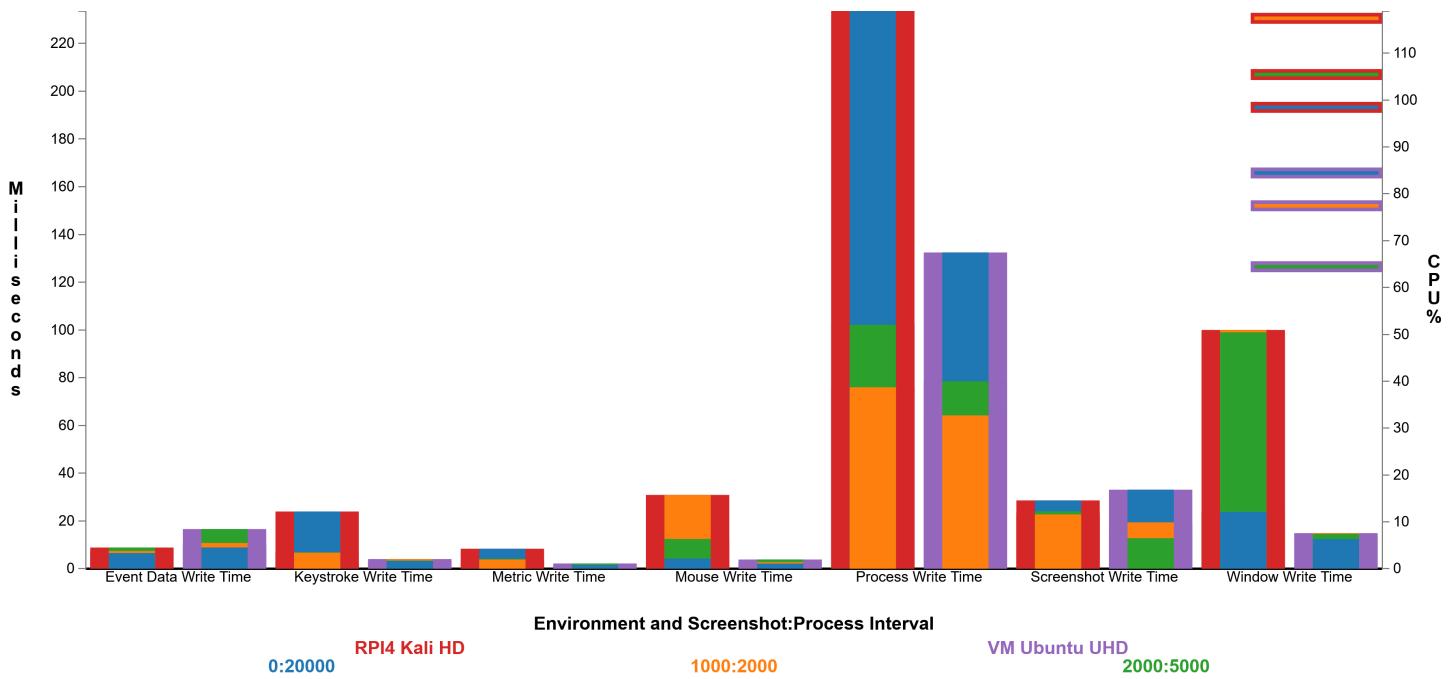


Figure 7.13: The average total time required to write each data category to disk per writeout. Note that averages are calculated only when entries to write exist; in cases like Process data larger intervals could lead to writeout periods where no data exists to write, leading to averages skewed to larger but less frequent writes.

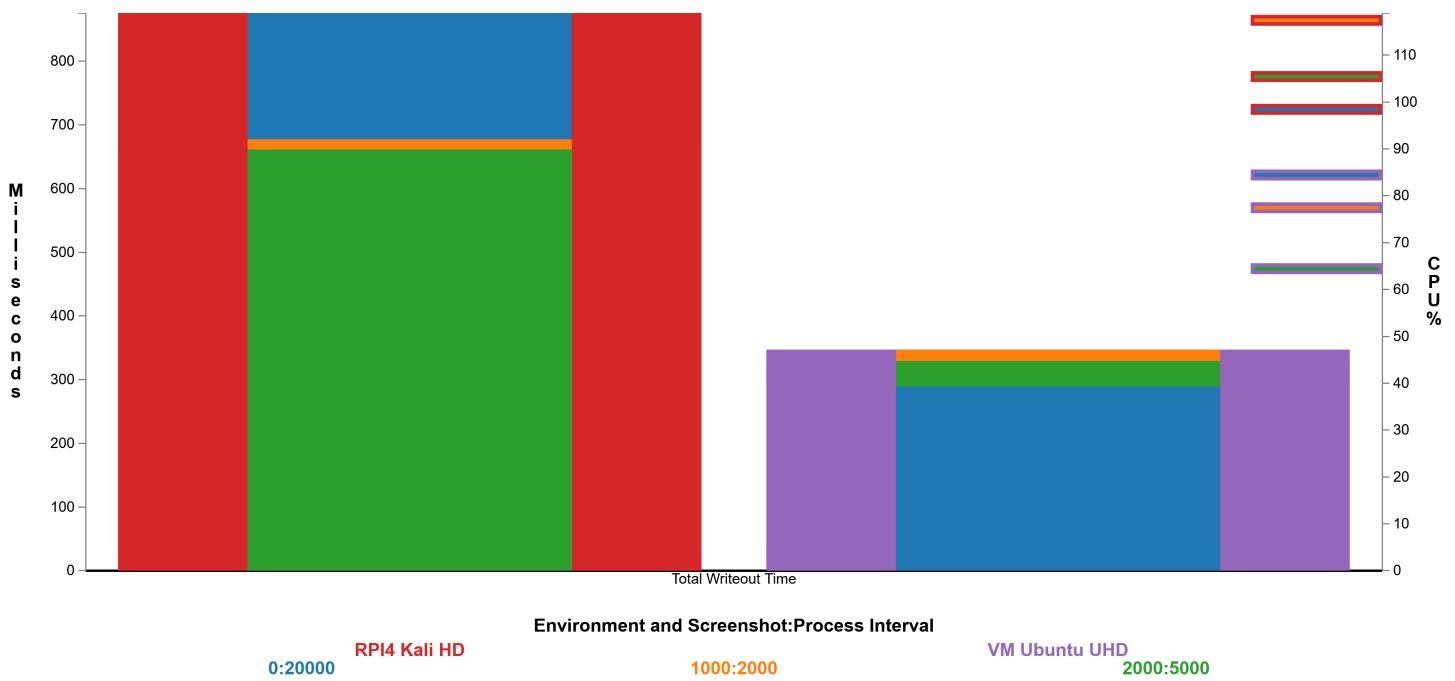


Figure 7.14: The average time taken to write a period's worth of data to local disk, demonstrating the superior persistence performance of the VM and interesting scaling characteristics.

Arch	CPU%	Mem%	FPS	Process	Screenshot
Kali RPI	100	4	1.201698514	20000	0
Kali RPI	119	4	0.841975309	2000	1000
Kali RPI	107	3	0.679775281	5000	2000
Ubuntu VM	86	15	2.434673367	20000	0
Ubuntu VM	79	9	0.795969773	2000	1000
Ubuntu VM	66	7	0.473972603	5000	2000

Table 7.1: Summary performance statistics running the endpoint monitor, for Kali RPI and Ubuntu VM. To ensure a uniform workload for the different tests we watched a roughly 5 minute YouTube video.

7.5 Summary

Our ultimate question is, “At what rate can the endpoint monitor unobtrusively collect data?” Table 7.1 shows a summary of our performance investigation. The testing demonstrated that over 2 screenshot frames per second could be achieved with reasonable (<90% of a single CPU load) performance. Higher frames per second may be achievable in systems with faster display access, such as bare-metal (as compared to virtualized) devices, or with lower display resolution.

Chapter 8

FUTURE WORK

Catalyst is under active development. Our goal is to continuously run events and collect data to get a complete understanding of the state of the art in reverse engineering and software protection, and to monitor how they evolve over time. We will continue to improve performance and platform independence to allow any interested subjects to run the system, regardless of platform. We will also improve the visualization component to make the annotation more efficient.

The most important part of future work, however, falls in three areas. First of all, an important realization that came out of this work is that, in order for the approach to software protection evaluation that we advocate for here to be truly useful, particularly as event participation grows, annotation of the collected data must be (at least) semi-automated. We discuss this in Section 8.1. Second, so far, we have not attempted to validate the generated models – do they, in fact, reflect the intent and high-level actions of the subjects? We discuss this in Section 8.2. Finally, in Section 8.3 we discuss the evaluation of the visualization component of Catalyst.

8.1 Speeding up Annotation

Research question 4 asks “Can the resulting datasets be lifted to human-understandable models of subjects’ behavior?” What we have found is that while the system is able to collect vast amounts of data from subjects, and we can generate Petri Net models from annotations, manually annotating the data requires a great deal of manual human labor. At the time of this writing, the research team annotated only several sessions in The Grand Re dataset constituting roughly 5 hours of trace. W found

that roughly each minute spent by subjects solving problems requires a minute of analyst annotation time. This is not sustainable for the larger competitions we envision for the future.

In this section we will discuss ways to resolve these issues: preprocessing of the collected data (Section 8.1.1), crowdsourcing annotations (Section 8.1.2), and, maybe most promising, using methods like machine learning and semi-supervised annotation (Section 8.1.3) to boost annotation productivity.

8.1.1 Preprocessing Screenshots

Much of the useful data collected is in the screenshots that are taken every couple of seconds. Unfortunately, while they contain a wealth of data, it is in a form that, by itself, is not amenable to further analysis. For example, we have found that the annotator often would like to search for “the next interesting event”, or “that point where the subject is shifting to using Ida,” etc. and such functionality is currently not possible. Essentially, the annotator would become more productive if they could search over the data stream.

8.1.1.1 Widget Extraction

Since the endpoint monitor collects all keystrokes, analyzing the actions of command line tools is straight-forward. If, for example, the subject types the step command in gdb, we will see the characters $\langle s, t, e, p \rangle$. Tools like Ida and Ghidra that are accessed through complex graphical user interfaces are a different story, however: when a user clicks on the **STEP**-button, all we see are the $\langle x, y \rangle$ -coordinates of the mouse click. In order to know that the subject was actually stepping forward in the program, those coordinates need to be analyzed in accordance with a user interface model of the application and view. Analyzing screenshots and detecting UI

widgets may be able to build such models.

Prior work demonstrates that user interface widgets can be detected in screenshots with varying levels of accuracy using image classification methods [85, 242]. We will build on such methods, but there are complications. While most subjects will use standard tools like Ida and Ghidra with known relative widget locations, advanced users will build their own tooling whose graphical user interface will be unknown to the system.

8.1.1.2 Video Segmentation

Knowing major event boundaries can speed up analysis. For example, highlighting areas where activity changes significantly and being able to fast forward between events such as “starting/stopping Ghidra”, “opening/closing emacs”, etc, can give the analyst a quick overview of the subject’s attack strategy. For this to be possible, we need to be able to segment the screenshot stream into ranges where the subject is likely to conduct the same activity.

User interface widget classification enables effective video segmentation as well, [205, 27] as discussed in Section 2.6.2.2. Video segmentation for screenshot data involves slicing videos into multiple parts based on the activity on screen. Prior efforts slice when a significant number of user interface cues change from one frame to the next, showing that activity on screen has changed. Widget extraction (through automated and semi-automated means, in prior work) aids in detecting these changes.

8.1.1.3 Text Extraction

Extracting text from screenshots may not be effective in annotating data, as it is likely redundant given that analysts can already see it in the visualization. However, this text may be useful to support search features or as additional features for semi or unsupervised learning (see Section 8.1.3). Additionally, screenshot analysis (particu-

larly text extraction) can be effective for information retrieval [353] tasks, which could aid analysts in finding sections of interest. Prior work demonstrates optical character recognition (OCR) efficacy in analyzing screenshots to extract text [181, 28].

8.1.1.4 Keystroke Lifting

In the screencast view, keyboard input is represented as text in the textbox. The research team, while annotating, found themselves looking up common hotkey combinations for different pieces of software users ran, such as the Ida Pro hotkey for stepping through a program. Instead of showing plaintext, it may prove helpful to replace these hotkey combinations with higher-level understandings, such as replacing the “F8” key with “Step Over” while users run Ida.

8.1.2 Crowdsourcing Annotations

Other work involving screencast annotation makes significant use of crowdsourcing [162]. Some of these efforts, such as using the Amazon Mechanical Turk, likely do not work well in the context of Catalyst analysis of The Grand Re datasets, as these traces contain complicated high-level labels requiring domain expertise to extract [251]. Nonetheless, crowdsourcing experts to annotate the data may be an effective strategy that can produce faster and more easily validated results than the small research team.

8.1.3 Machine Learning and Semi-supervised Annotation

In The Grand Re datasets, analysts in small scale evaluation annotated datasets in approximately real time: For each minute a subject spent conducting tasks during a trace, analysts spent roughly one minute in analysis. While this may be feasible in annotating The Grand Re dataset, larger data sets will likely require more automated annotation through semi-supervised methods. A significant amount of work applies

machine learning methods to high-dimensional time-series data. [6, 51, 306, 20, 362, 354] These methods — which have not yet been applied to the HDI annotating problem domain — attempt to annotate data and forecast upcoming data. Using the data sources (including annotations) as feature vectors, these methods could potentially aid in generating annotations: Algorithms can propose annotations based on the underlying data or on prior annotations, or a combination of both.

Already, researchers employed semi-supervised methods in annotating screencast videos with some auxiliary data like mouse and keyboard input, and those methods specifically might be leveraged here [162]. That related work, in particular, utilized a semi-supervised annotate-on-low-confidence method for some types of analysis; that model can potentially be used here by building automated classifiers and having humans evaluate segments on session timelines that are not easily or confidently classified.

8.2 Validating Generated Models

Currently, the quality of generated annotations and derived Petri nets have yet to be formally evaluated, though informal evaluation from The Grand Re research team is discussed in Section 6.2. A number of interesting experiments can be conducted to perform such validation.

Most interesting, perhaps, is to conduct post-event interviews with subjects where they are asked to evaluate the generated Petri nets: do they, in fact, correspond to the attack strategies followed by the subjects? Such interviews have to be completed soon after the submission of a solution and before the subjects' memory of the event begins to fade. A variant of such interviews would have the subjects view the raw annotations, rather than the derived Petri nets.

It was not possible for us to conduct post-event interviews for the *Grand Re* since,

at that point, the Petri net generation algorithm was not yet fully implemented.

In a second experiment subjects could be asked to generate their own annotations and compare those to annotations done by analysts. The endpoint monitor has facilities for allowing the subject annotate their actions as they perform them, although few have availed themselves of this possibility¹.

In a third experiment, experienced reverse engineers would be given a challenge and a Petri net derived from a subject's solution to that challenge. The reverse engineer would then be asked to reproduce the attack. If unsuccessful, it can be surmised that some important step was left out of the model.

8.3 Evaluating Visualizations

While the Catalyst visualization and annotation tool has been successfully used to annotate traces by myself and researchers from the Ghent University team, there has been no formal evaluation as to how well this visualization works. It could well be that a more intuitive visualization would allow for a more efficient annotation process. Studying subjects as they annotate would provide understanding and insights into how well the different components perform in practice. We are considering several experiments, described below.

In a first experiment, components could be selectively removed from the interface (or added as new ones become available) and the difference in annotation time could be measured.

In a second experiment, rather than being asked to annotate a complete trace, annotators could be asked to perform specific tasks such as "find active windows," "synchronize annotation timestamps with other data points," "identify important

¹In the *Grand Re* an additional 1/2 point was awarded to solutions submitted with good subject annotations, but no one was thus rewarded.

background processes,” or “trace user input” and the time to do so would be measured.

Finally, in a third experiment, multiple subjects could be monitored as they annotate a particular trace to examine the relative time spent in different components.

Chapter 9

CONCLUSION

We recently received a request from the founder of a company that builds software protection tools for JavaScript:

“What our customers really want is a button that they can press that will tell them how well the obfuscating transformations they have selected will protect their web applications from attack.”

While such a facility will likely never be realized, the methodology put forth in this dissertation has allowed us to collect data from accomplished reverse engineers attacking applications protected with state-of-the-art academic and commercial tools, and derive attack models from this data. Ultimately, my hope is that one day these models will assist users of software protection tools better select the right protection for their application and attack scenario.

In this chapter I will conclude by revisiting the research questions from Section 1.2.

9.1 Research Question 1

RQ1: Is it possible to build a system to remotely and unobtrusively collect high fidelity HDI data from experts in their own dissimilar computing environments as they reverse engineer software?

The Catalyst Data Collection Engine demonstrates a tool that remotely collects high fidelity HDI data. I built this tool with performance and cross-platform support in mind and it served its purpose well in reverse engineering challenge events.

Catalyst is designed to minimize the hardware and connection capabilities required to run the front-end data collection on subject devices. Catalyst consumes about a single x86 processor core and on the order of 2GB of memory to operate; in RevEngE and The Grand Re, subjects were able to generate traces on their hardware with this level of overhead. In small scale, informal testing, Catalyst even ran on a Raspberry PI 4 without issue.

Because Catalyst sources common data¹ from operating systems using cross-platform libraries and stores and presents that data in a uniform way, it supports dissimilar computing environments.

Our experiments demonstrate that it is possible to remotely collect high fidelity HDI data: Reverse engineers used it in dissimilar computing environments to produce traces for RevEngE and The Grand Re.

9.2 Research Question 2

RQ2: What type of incentives are needed to recruit significant number of expert reverse engineer participants running this system?

In RevEngE and The Grand Re, subjects who completed reverse engineering tasks won monetary prizes: \$1,500 in the former and \$7,000 in the latter. The \$1,500 award generated about 25 hours of reverse engineering time, while the \$7,000 “bought” just over 100 hours of reverse engineering time from two individuals. On average, successful participants cost \$68 per hour of trace data — \$60 per hour for the RevEngE studies and \$70 per hour for The Grand Re competition.

Money alone, however, is not the only factor to consider when conducting studies

¹Common refers to the cross-platform nature of the data here; all supported operating systems contain GUIs displayed on monitors with windowing and process management systems, as well as standard keyboard and mouse input.

like these, as earlier rounds of RevEngE offered similar prize amounts and did not receive much participation. Rather, it is important to appropriately advertise the studies, and in that effort it appears that a larger research team with recognizable prior work in reverse engineering provides the credibility necessary to encourage participation. Additionally, the structure of the competition and studies likely make a difference, based on the results from the 4 events here. Competitions with a large number of challenges did not generate as much interest as smaller scale, more focused events. The individual, guaranteed reward study also generated some high quality data, at a lower price point than the competition format. The study format also, however, ultimately produced less data than the competition format.

We are planning to continue running Grand Re-like events, and it will be interesting to see if, over time, as the event becomes better known in the reverse engineering community, participation improves.

9.3 Research Question 3

RQ3: Will generated synthetic reverse engineering challenges capture real-world reverse engineering tasks?

The studies described here contained challenges generated by a research team consisting of academics and industry experts who study and deploy code obfuscation, both in the abstract and in practice. This team generated the challenges based on real-world use cases found in related work, as discussed in Section 2.1. Moreover, expert reverse engineers from similar fields successfully solved these challenges in relatively short amounts of time ranging from on the order of several hours (in the RevEngE Study) to about 2 months (in the case of The Grand Re) using what, at first analysis, appear to be typical, well-known reverse engineering methods. All of

this suggests that they had encountered similar problems before.

Despite these factors suggesting that the challenges mirror real-world tasks, other differences remain. In the real world, software that reverse engineers target often exceeds the challenge sizes by orders of magnitude. The amount of functionality for real-world software likewise exceeds the small-scale synthetic challenges which focus on a single functionality. However, while real-world applications may be orders of magnitude larger, their *security kernels* (that need to be protected with obfuscation and tamper-proofing) may well be on the order of the size of our challenges. To know whether this is, in fact the case, it would be necessary to reverse engineer and analyze commercial software protected with state-of-the-art protection tools, which is beyond the scope of this thesis.

9.4 Research Question 4

RQ4: Can the resulting datasets be lifted to human-understandable models of subjects' behavior?

I developed visualization and annotation tools for HDI data. The Grand Re research team and I applied these to that event's dataset, and I also annotated RevEngE-based datasets. In all, we found that these tools did enable us to annotate HDI data in the context of reverse engineering data using these tools; the Petri net generator algorithm I developed also allowed us to lift these annotations to human-understandable models.

Therefore, we find that it is possible to lift HDI traces to human-understandable models. However, there are some issues with the current methods as well.

Most seriously, two issues have conspired to make annotation difficult: even with with a small number of subjects many hours of traces were collected, and the anno-

tation process so far has proven to be slow (roughly real-time: 1 minute of analysis per 1 minute of trace). This makes annotating many subjects' data difficult with a small research team. Furthermore, because of the ambiguity that can arise in annotation, multiple researchers are needed to annotate each trace and, when conflicts arise, there needs to be a way to resolve them. This leads to even higher annotation costs. As discussed in Section 8.1, the annotation process needs to be streamlined by (semi-)automatically extracting information from the collected low-level data and adding semi-supervised automation. Alternatively, simpler challenges could be used, resulting in shorter and more easily annotated traces, but this removes the challenges even further from real protected applications.

Further issues relate to the validation of the Catalyst toolchain. In particular, the visualization tool has not been formally evaluated to determine how effective it is in helping researchers with their annotation tasks.

Finally, the resulting attack models need to be validated to determine how close they are to the tools and tactics actually used by the subjects. This requires that researchers, post-events, to revisit the models with the subjects who are the ultimate arbiters of model validity.

9.5 Research Question 5

RQ5: Do these analyses and models provide valuable insight into reverse engineering practices and code obfuscation efficacy?

Thus far, only the RevEngE and Grand Re research teams have evaluated the current annotations and resulting Petri net models. We believe that these do provide some insight into reverse engineering practices — we observe, for instance, subjects using dynamic analysis techniques to extract assets from all three challenge types

included in the studies; subjects only solved challenges with what the research teams believed to be light or (at most) medium intensity protection, and the traces lead us to believe that the subjects' methods involved known and documented techniques and tools. Finally and independently of the HDI data, the challenges demonstrate that man-at-the-end attackers can overcome virtualization, opaque predicate, function flattening, encoded literals, and self modifying code obfuscations coupled with checksum and code guard protections, all within a reasonable amount of time — with no challenge taking more than 50 hours at most. The HDI data, once fully analyzed, will likely offer more specific timeframes for each problem, providing insight to the obfuscations and protections used.

By viewing the models generated from the annotation we are able to see subjects' methods step by step, though the specificity of the steps depends on the specificity in the annotations. More work with regard to the Catalyst toolchain is needed to automatically integrate underlying data sources (such as active windows, processes, and resource use) into the models — these sources can add specificity where the annotations derived by manual labor may not.

To be answered conclusively, this question requires more raw data from more challenges, more annotations of the underlying data, and multiple annotations from different analysts for each trace. It is encouraging, however, that for the first time the community has collected high-quality data of *real* reverse engineers in the wild, attacking challenges protected with *real* (i.e. used not only in academia but also in industry) software protection tools. We believe that this means means that there is finally real hope of answering the question that has been vexed the field since its inception: how well do obfuscating transformations stand up to concerted attacks from well-trained reverse engineers?

9.6 Released Data

The data generated from RevEngE and The Grand Re is publicly available in anonymized form. Anyone may view this data using the Chrome browser at:

<http://revenge.cs.arizona.edu/DataCollectorServer/openDataCollection/vissplash.jsp?event=GrandReChallenge&eventPassword=anondata&eventAdmin=cgtboy1988@yahoo.com>

The raw data can be downloaded in a JSON format. Additionally, an archived anonymized version of the underlying MySQL formatted database is available here:

<https://doi.org/10.25422/azu.data.19858645>

It is my hope that the computer security community will find Catalyst a useful tool for further attack data collection, and that the software protection and reverse engineering communities will find the collected data useful for further study.

Appendix A

CATALYST DATA DICTIONARY

Catalyst Data Dictionary			
Note: The index field is used to determine the place of the data entry on the overall timeline. It is a duplicate of another piece of data and represents when the piece of data occurred. The data piece it is duplicated from has more detail regarding what it represents for each individual type of data.			
Data Category	Field	Definition	Index?
windows	Note: Fields in the "window" data category relate to the current active in focus window and attached processes. Window data is checked upon interrupt triggered by mouse or keyboard input.		
windows	ChangeTime	This is the index. Window information is recorded whenever that information changes, so the field is the time when the window comes into focus. This also denotes when the previous window went out of focus.	Yes
windows	User	Operating system user owning the window. This is part of the key to connect to data of the "process" category.	
windows	Start	The start time of the attached process, see ps command in linux for details on formatting. This is part of the key to connect to data of the "process" category.	
windows	PID	The PID of the process attached to the window. This is part of the key to connect to data of the "process" category.	
windows	Name	The title of the window.	
windows	FirstClass	See X11 WM_CLASS documentation.	
windows	SecondClass	See X11 WM_CLASS documentation.	
windows	XID	The X ID of the window. See X windowing system for details.	
windows	X	The x coordinate of the window.	
windows	Y	The y coordinate of the window.	
windows	Width	The width of the window.	
windows	Height	The height of the window.	
processes	Note: Fields in the "processes" data category relate to the current processes running both in the foreground and background. Process data is polled and upon interrupt if the active window changes.		
processes	SnapTime	This is the snapshot time, the time at which the sample was taken.	Yes
processes	User	Operating system user owning the process.	
processes	Start	The start time of the process, see ps command in linux for details on formatting. Note that, because the time switches to date after active for a day, the keying based on this data point may need careful examination.	
processes	PID	The PID of the process.	
processes	Command	The command issued to start the process.	
processes	Arguments	The arguments issued with the command.	
processes	CPU	The CPU usage at the snap time, see ps command in linux for details.	
processes	Mem	The memory usage at the snap time as a percent of available system memory, see ps command in linux for details.	
processes	VSZ	The memory allocated to the process.	
processes	RSS	The memory used by the process at the snap time.	
processes	TTY	The terminal attached to the process.	
processes	Stat	The process state, see ps command in linux for more details.	
processes	Time	The amount of CPU time the process has used.	
events	Note: Tasks are annotations of what a user is doing on the device. Events are components of these tasks, such as "start", denoting the start of a given task, and "end". Other events may occur, but these are the types initially supported by the data collection software.		
events	EventTime	The time when the given event incident happened.	Yes
events	TaskName	The name of the task that the user is doing.	
events	Completion	A boolean which is true if the event was completed at some point.	
events	Description	The description of the event, such as "start" or "end".	
screenshots	Note: Screenshots are taken on a polling basis and additionally are taken when a window change is detected. The polling duration is adjusted for performance depending on OS/device type.		
screenshots	Taken	The time when the screenshot was taken.	Yes
screenshots	Screenshot	The screenshot itself, in base 64 for JSON-only exports.	
screenshots	Path	The relative path to the image in a ZIP export.	
keystrokes	Note: Keystrokes are recorded as they are pressed. These entries connect to the active window at the time.		
keystrokes	InputTime	The timestamp for when the button was pressed.	Yes
keystrokes	TimeChanged	The ChangeTime for the active window at the time (see "windows" above).	
keystrokes	XID	The X ID of the active window at the time (see "windows" above).	
keystrokes	Start	The Start field of the process attached to the active window (see "windows", "processes").	

The data dictionary lists the attribute fields of the collected data points, organized by data type.

keystrokes	PID	The process ID (PID) field of the process attached to the active window (see "windows", "processes").	
keystrokes	User	The operating system User owning the active window and associated process (see "windows", "processes") at the time.	
keystrokes	Button	The button pushed (such as "A", "B", etc.) by the user.	
keystrokes	Type	The type of keystroke event (such as "pressed", "released", etc.).	
mouse	Note: Mouse inputs are recorded as they are pressed. These entries connect to the active window at the time.		
mouse	InputTime	The timestamp for when the button was pressed.	Yes
mouse	TimeChanged	The ChangeTime for the active window at the time (see "windows" above).	
mouse	XID	The X ID of the active window at the time (see "windows" above).	
mouse	Start	The Start field of the process attached to the active window (see "windows", "processes").	
mouse	PID	The process ID (PID) field of the process attached to the active window (see "windows", "processes").	
mouse	User	The operating system User owning the active window and associated process (see "windows", "processes") at the time.	
mouse	XLoc	The x location of the mouse at this event.	
mouse	YLoc	The y location of the mouse at this event.	
mouse	Type	The input type (such as "up", "down", etc.).	

Appendix B

CURRENT IRB APPROVAL



THE UNIVERSITY OF ARIZONA

Human Subjects
Protection Program

1618 E. Helen St.
P.O.Box 245137
Tucson, AZ 85724-5137
Tel: (520) 626-6721
<http://rgw.arizona.edu/compliance/home>

Date:	March 05, 2020
Principal Investigator:	Clark Garrett Taylor
Protocol Number:	1610963521A002
Protocol Title:	Building Attack Trees: The Best Tools for Reverse Engineering
Level of Review:	Expedited
Determination:	Approved
Expiration Date:	November 22, 2022
Change Description:	Currently, this research provides financial incentives for participants who successfully complete particular challenges hosted on a web application. In addition to hosting challenges which have a specific financial incentive for completion, we wish to host temporary, three day to four week long competitions with a financial incentive for participants who perform the best (see attached draft schedule).
Documents Reviewed Concurrently:	
HSPP Forms/Correspondence: <i>amendment_approved_research_v2019-06.pdf</i> HSPP Forms/Correspondence: <i>f200_v2016-07_final_ctf_update.doc</i> Informed Consent/PHI Forms: <i>ICF Professionals_updated.doc</i> Informed Consent/PHI Forms: <i>ICF Professionals_updated.pdf</i> Informed Consent/PHI Forms: <i>ICF Students_updated.doc</i> Informed Consent/PHI Forms: <i>ICF Students_updated.pdf</i> Participant Material: <i>RevEngE Schedule.pdf</i>	
Regulatory Determinations/Comments:	
<ul style="list-style-type: none"> • Modification Eligible for Expedite Review (45 CFR 46.110): The modification(s) do not affect the design of the research AND the modification(s) add no more than minimal risk to subjects. 	

This project has been reviewed and approved by an IRB Chair or designee.

- No changes to a project may be made prior to IRB approval except to eliminate apparent immediate hazard to subjects.
- The University of Arizona maintains a Federalwide Assurance with the Office for Human Research Protections (FWA #00004218).
- All research procedures should be conducted according to the approved protocol and the policies and guidance of the IRB.
- The current consent with the IRB approval stamp must be used to consent subjects.
- The Principal Investigator should notify the IRB immediately of any proposed changes that affect the protocol and report any unanticipated problems involving risks to participants or others. Please refer to Guidance *Investigators Responsibility after IRB Approval* and *Reporting Local Information*.

Appendix C

STUDENT CONSENT DOCUMENT



The University of Arizona Consent to Participate in Research

Study Title: Building Attack Trees: The Best Tools for Reverse Engineering

Principal Investigator: Clark Taylor

Sponsor: National Science Foundation

Summary of the Research

The researchers here seek to understand better the process by which humans reverse engineer computer code, particularly when that code has been obfuscated. Specifically, the researchers are interested in what the best reverse engineering practices are for given obfuscation types and how effective those strategies (and the opposing obfuscations) are. To do that, the researchers are hosting obfuscated code challenges which participants reverse engineer while using data collection software. That data is then to be analyzed in order to answer the research questions.

The researchers intend this study to be an ongoing effort, with participants investing time as they see fit. As participants solve difficult challenges, they can earn monetary rewards for their participation. There is little associated risk, as participants simply work on problems from their computer or device at their convenience.

This is a consent form for research participation. It contains important information about this study and what to expect if you decide to participate. Please consider the information carefully. Feel free to discuss the study with your friends and family and to ask questions before making your decision whether or not to participate.

Why is this study being done?

This study focuses on how professionals reverse engineer computer code. Data collected from the subjects of the study will be aggregated in order to build attack trees, which may serve as guides in solving reverse engineering problems.

What will happen if I take part in this study?

Students at the University of Arizona will have the option of taking part in this study while completing any computer security/reverse engineering/compiler courses. During the survey, data will be collected from you while you solve the reverse engineering problems provided by the investigators on their website. The data will be collected on the virtual machine provided on that website using software preinstalled on the virtual machine. The software will only be collected once activated by scripts, as detailed on the site. The data will only be uploaded upon

HSPP Use Only:
 Consent Form T502a v 2016-07

Consent Version: 11/22/2016

Page 1 of 6

There also exists an almost identical version of this document for professional participants.



submission of a problem solution, using the submission script. You will be prompted with the option to submit the data at that time.

How long will I be in the study?

You will be in the study as long as you wish to continue solving the problems on the website while collecting and submitting data. All data collected and submitted will be done by you through the provided scripts. Thus, you are in the study only as long as you wish to be in the study and simply refusing to collect and/or upload data will cease your participation in the study. Data already collected before ceasing activity will remain in the study. You have the option to solve problems without collecting data as well.

How many people will take part in this study?

The investigators cannot easily estimate the number of individuals who will take part in the study, as participation is entirely voluntary and available to anyone who signs up on the web application. The web application used for the study will be scalable to hundreds to thousands of users; the investigators doubt that the number of subjects will exceed 100 professionals participating who are not involved in a course. Additionally, the investigators expect to have a maximum of about 20 to 80 students participating in the study each semester while taking a University of Arizona security course.

Can I stop being in the study?

Your participation is voluntary. You may refuse to participate in this study. If you decide to take part in the study, you may leave the study at any time. Data will only be collected upon your running and activating it in the provided script; data will only be uploaded to the investigators when you choose to do so in the submission script. No matter what decision you make, there will be no penalty to you and you will not lose any of your usual benefits. Your decision will not affect your future relationship with The University of Arizona. If you are a student or employee at the University of Arizona, your decision will not affect your grades or employment status.

What risks or benefits can I expect from being in the study?

There are few risks associated with this study. The investigators have taken reasonable security measures on the infrastructure (the web application and virtual machine provided) but cannot guarantee that the data collected will be completely secure; a security breach may possibly render it vulnerable. This information includes email addresses and names, though you are free to provide fake email addresses and/or fake names if desired. Passwords will be salted and hashed, but please do not use a password that you use in other applications.

Will I receive compensation for being in the study?

There is no general compensation for solely participating in this study. However, there may be compensation for solving certain types of reverse engineering problems. Throughout the

[HSPP Use Only:
Consent Form T502a v 2016-07]

Consent Version: 11/22/2016

Page 2 of 6

Appendix D

GRAND RE CHALLENGE DATA SUMMARY

User	Session	Total Time	Active Time	Task	Window	Process	Screen	Mouse Input	Key Input
User19	...3072bd1eba	0	5	0	1	234	2	15	0
User19	...0c70d5361c	196	135	16	3860	143462	1575	3582	6567
User19	...0e0bbec076	2	10	1	18	1409	17	123	12
User19	...72786e801c	123	15	4	19	6359	45	575	507
User19	...4d68280869	42	45	3	344	28848	246	2397	2691
User19	...dcceabc5c5	0	5	0	6	267	5	15	0
User19	...f00f1a9842	184	20	1	45	117837	752	197	117
User19	...bd83d43be3	517	210	51	5069	170654	2298	5724	12071
User19	...917e8cc478	500	315	7	5971	360061	3211	9839	19772
User19	...7db5616821	1308	90	5	2855	91165	1001	1395	5833
User19	...9902edb305	2569	135	2	2257	251771	1962	5168	10073
User19	...142a66c3ff	11	15	0	312	8175	133	388	1812
User19	...1190c049d2	182	115	0	2254	135434	1158	2598	4574
User19	...2df205f932	29	30	0	762	20415	271	1060	2089
User19	...a800a53659	625	260	1	5267	466949	3375	8453	12537
User19	...645fd7fd62	4	10	1	68	2707	41	192	63
User19	...76e1c94950	17	25	0	395	12177	163	743	1628
User19	...454754915f	47	50	1	457	32765	289	1815	1879
User19	...ea3b16b029	654	305	1	7606	487844	4138	8191	12927
User19	...61c1fb2b55	81	75	1	1942	61113	766	2419	3360
User19	...63b86abd52	250	190	1	4857	193825	1856	5322	10887

User19	...c4330b7ed8	191	155	1	2235	136516	1277	3803	6064
User19	...99f2b55c9f	2009	30	2	78	21781	151	595	191
User19	...845d22a298	61	65	1	1374	45096	564	1872	3048
User19	...3ff8a34bed	1459	125	6	1311	325756	2172	3011	9528
User19	...a54508d0c2	105	60	0	350	74786	800	1487	2654
User19	...b48d151b05	552	55	1	505	387508	2580	839	918
User19	...9a9e62947b	260	35	1	425	162655	1009	319	1803
User19	...de7573f579	283	55	3	1403	199164	1459	1927	3364
User19	...8ee4a62bbc	2	5	0	59	1468	31	70	6
User19	...556a5fbbe5	284	250	1	5316	209300	2138	5245	22602
User19	...1dc68604cb	276	205	0	2222	198534	1626	2040	6300
User19	...83fba4c33c	310	135	2	1622	217359	1693	3431	6618
User0	...63287cd946	2	10	0	45	588	2	6	0
User0	...88c69bdd40	56	0	0	5	16412	194	0	0
User0	...34fe8a1233	7	15	0	97	2123	8	19	6
User0	...8a0e101297	4	5	0	9	1277	21	2	0
User0	...9f5bd4d6f6	7	10	0	20	1840	10	23	0
User0	...9656fdac7	1	0	0	5	133	3	0	0
User7	...14c568ba9b	3	10	2	2	2001	15	32	183
User7	...197e75d98d	16	15	2	17	9500	72	10	81
User7	...2aeafba47e	5	5	1	1	3861	23	4	78
User7	...976d1c24c8	18	10	4	594	1383	13	23	227
User7	...968782b565	1	5	1	2	1279	8	4	87
User122	...ddeb89ce49	124	70	1	5	38481	311	497	2913
User122	...a04c27ee25	767	80	0	5	376309	3069	204	2343
User122	...7263e6aeeb	41	30	0	5	20497	166	243	2103
User122	...6511eee5f7	792	175	0	15	393477	3172	773	3971
User122	...bc59beead1	3212	400	0	63	1608595	12856	2593	14622
User101	...b9eec0412d	5083	165	12	4	91731	597	872	22844

User101	...	20bf9b3d83	4342	400	4	7	2599169	15093	1559	46656
User95	...	eefa01989e	888	10	2	24	305650	3497	22	24
User95	...	e901a5a0b3	3	0	0	5	1515	16	0	0
User127	...	f7e8023824	97	50	2	77	3178	453	296	4501
User127	...	93039b7f27	263	40	18	261	89223	741	395	1351
User127	...	f2df552dd7	204	10	0	44	115107	841	70	2765
User97	...	2458893cec	0	5	0	5	266	4	8	1
User105	...	b7ff73e6eb	832	145	2	9	441637	3331	1362	8933
User105	...	141d1f49ed	1262	40	0	2	655485	5049	124	333
User98	...	774cb294dc	492	295	0	33	222559	1390	3513	27230
User103	...	401f986def	21	25	0	39	1768	4	11	0
User103	...	49e57412ef	128	40	13	270	62153	612	452	899
User99	...	06773764d5	4016	10	0	3	3486780	16065	32	30
User99	...	5d83025dac	2790	50	0	90	50746	250	215	3063
User56	...	0c0d850e92	17	20	0	5	8090	70	320	3268
User56	...	b849b45df1	86	35	2	64	11003	96	295	1069
User56	...	eaec81f506	0	5	0	4	155	3	18	42
User12	...	ef5c1b2021	5	10	0	62	2393	52	81	21
User12	...	1228ffe92b	0	5	0	1	175	1	2	0
User12	...	accce275a2	1	5	0	4	346	6	12	0
User12	...	0fbf45f292	2	10	0	25	1102	20	20	39
User12	...	271be10c17	0	5	0	2	175	2	3	0
User12	...	72c42e2670	1	5	0	4	705	7	24	6
User12	...	5b3cd35dc9	1	0	0	0	6	8	0	0
User109	...	eb90769a84	1	10	1	8	547	11	26	3
User107	...	4953a86a1b	10	15	1	14	8688	43	219	2175
User59	...	1f74ac2a66	35	40	0	381	30085	225	584	1265
User59	...	225c27a2b3	7	15	0	160	5775	62	98	0
User130	...	3c714f5a7f	47	45	3	48	1957	243	126	4339

User130	...aa8b2df392	102	5	0	1	62373	409	2	0
User130	...bb67b671c1	1506	35	3	186	54436	389	235	309
User130	...474ec4c677	714	5	0	1	429501	2844	4	0
User130	...39135aa23d	33	15	0	11	21863	143	52	66
User130	...bc23bca779	61	55	0	319	39799	326	921	2258
User130	...ec11048887	20	10	0	33	13066	92	76	492
User130	...b68b3147d0	35	25	0	111	21948	175	158	702
User111	...7acfbd7934	10	0	2	0	23	33	0	0
User111	...42c7317037	7	5	0	17	145	43	43	223
User111	...e0ce3e3786	148	30	4	64	63550	577	54	1328
User111	...060d6cf64d	18	20	1	7	427	78	22	1440
User61	...f2e654ea36	19	25	2	36	10600	83	364	3610
User62	...215faf905e	157	85	4	57	34572	322	1524	6236
User110	...9c9db49a02	2	5	0	9	1086	12	12	9
User63	...80b3b3929c	50	55	0	617	22116	56	155	2
User63	...588ceb3796	30	35	2	336	13464	33	47	39
User63	...79d296e565	21	25	2	291	9770	36	59	21
User63	...71cf536ad2	35	40	31	1471	17259	362	1502	2682
User116	...1fc838d3a1	3	10	4	2	1492	12	24	54
User116	...956896868e	972	5	6	2	176	2	10	0
User116	...0184987d7b	1216	360	0	22	630266	4869	4398	19370
User116	...2a22d2199b	7	5	0	2	3700	30	26	248
User116	...34fefafa234a	314	25	0	3	161764	1258	26	1451
User116	...1cdd149744	164	60	0	10	85408	660	1131	9228
User67	...6451be5bcf	1	5	0	10	980	10	12	21
User67	...6f620ed802	5	10	0	83	3643	45	71	159
User67	...cdd346de3a	762	25	1	138	9382	85	359	948
User67	...828d084f9d	0	5	0	3	256	2	3	0
User67	...aa3defe860	0	5	0	5	253	4	2	0

User	...	Time	Count	Count	Count	Count	Count	Count	Count
User67	...2a57156e4c	0	5	0	1	252	2	4	0
User67	...c78239a7eb	0	5	0	1	214	2	5	0
User67	...a7488145c7	0	0	0	1	240	1	0	0
User67	...dcc5f1651c	39425	25	2	140	18611	178	268	1879
User67	...8b67cfa4d6	0	5	0	1	222	2	1	0
User67	...6de1b7b62d	0	5	0	2	221	3	1	0
User67	...d99d5b2c32	0	5	0	2	244	3	6	0
User67	...f00a7483f7	0	5	0	1	221	2	1	0
User67	...80b39e52ac	0	5	0	1	234	2	2	0
User67	...b1136cf5ca	0	5	0	1	231	2	1	0
User67	...28fbef3fa5	0	5	0	2	234	4	17	0
User67	...84811bc072	0	5	0	1	216	2	1	0
User67	...cda0e4e24c	0	5	0	1	231	2	5	0
User67	...68532ec6eb	0	5	0	1	237	2	2	0
User67	...bdfc221957	0	5	0	1	240	2	1	0
User67	...b07e110e4c	0	5	0	1	228	2	1	0
User67	...ac0eaf106a	0	5	0	1	241	2	5	0
User67	...31bdb1f693	0	5	0	4	244	4	2	0
User67	...d83d67f309	38	0	0	1	28668	152	0	0
Total		84698	6680	249	67821	17013866	120898	111657	382741
Winners		15630	3430	78	61975	5218337	43425	86424	180661
		Total Time	Active Time	Task	Window	Process	Screen	Mouse Input	Key Input

Table D.1: A summary view of subjects' activity from the Grand Re Challenge. Time is measured in minutes, while other columns are data entry counts. Usernames are anonymized automatically in Catalyst in the public data view, available at <http://revenge.cs.arizona.edu/DataCollectorServer/openDataCollection/vissplash.js>?event=GrandReChallenge&eventPassword=anondata&eventAdmin=cgtboy1988@yahoo.com.

Appendix E

CATALYST ER DIAGRAMS

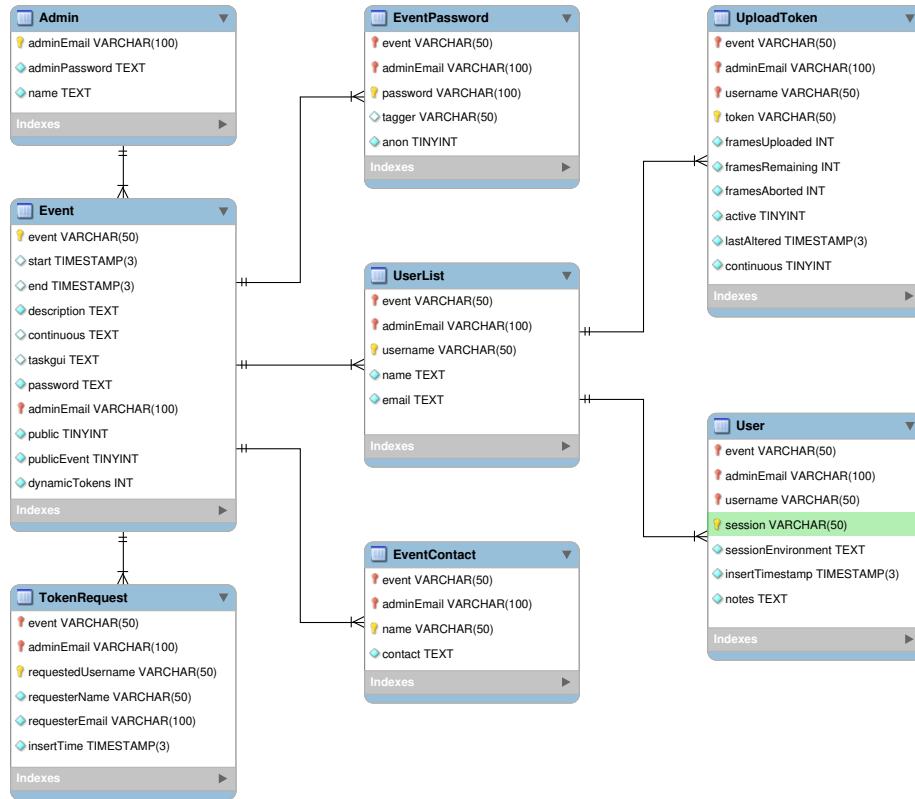


Figure E.1: The administrative tables for Catalyst. The ‘TokenRequest’ table contains requested tokens for events that do not automatically approve new tokens. ‘EventPassword’ contains authentication information for non-administrators to view and annotate the subjects’ data. The ‘UserList’ contains users’ authorized tokens, which serve as authentication to download an installer. Each installer download inserts a new ‘UploadToken’ and passes it to the installer, which authenticates data uploads from the Endpoint Monitor. Diagram produced with MySql Workbench. [204]

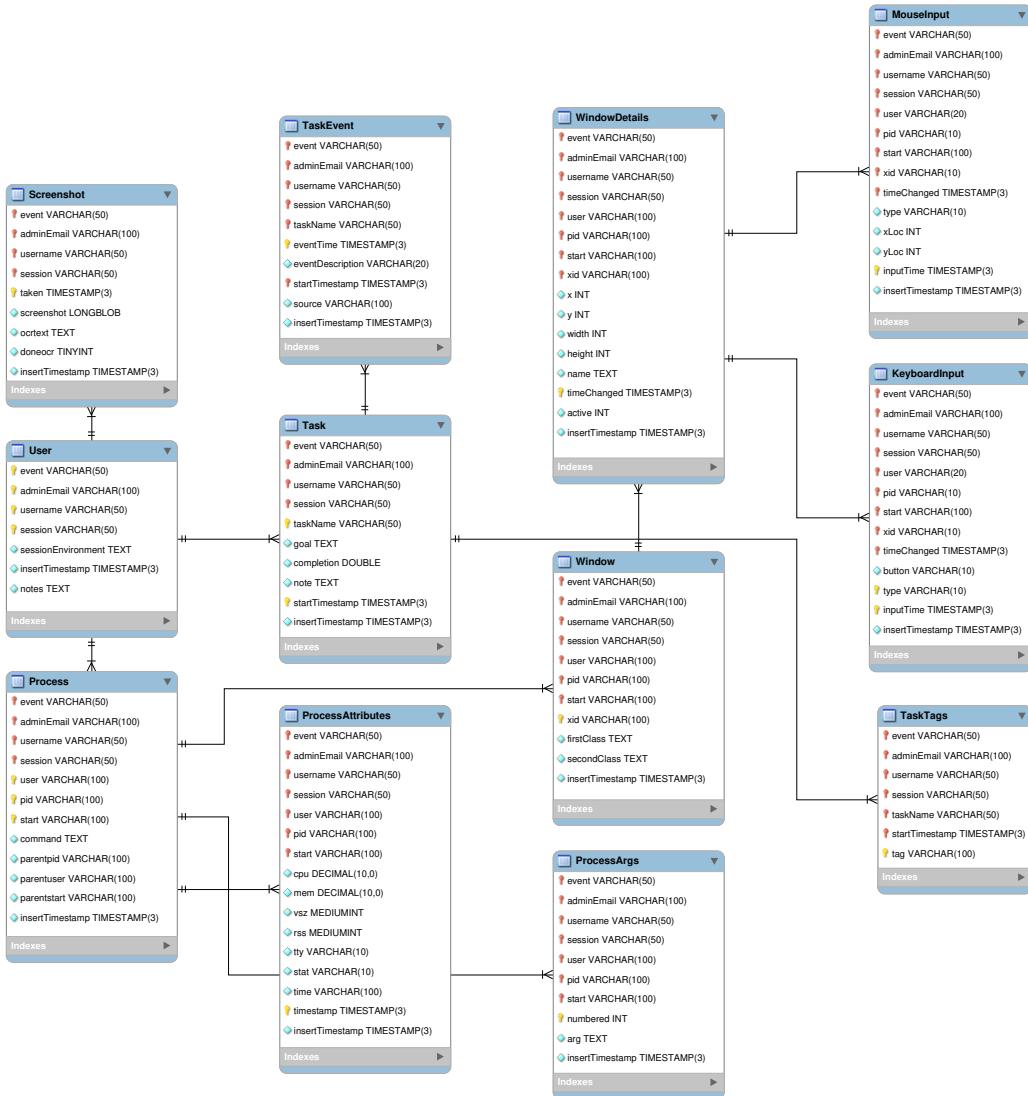


Figure E.2: Subject data tables key from the ‘User’ table. As Catalyst streams data from subjects, the different data types map to the tables here. Data exports generally operate on these tables, though some data statistics are cached as shown in E.3. These Endpoint Monitor operates largely on identical tables provisioned to a MySQL instance installed on subjects’ devices. Diagram produced with MySQL Workbench. [204]

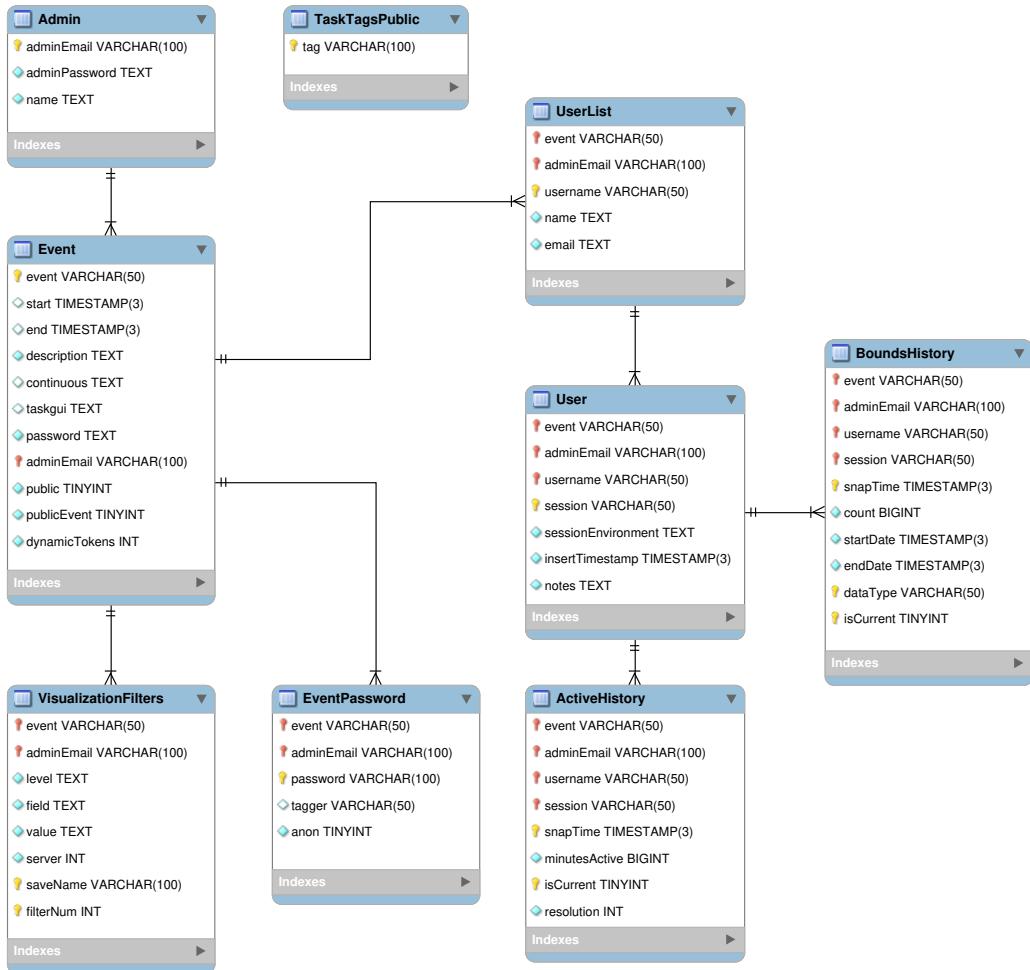


Figure E.3: Additional tables provide functionality primarily for the visualization. ‘VisualizationFilters’ stores filter sets so that they might be quickly applied; ‘BoundSHistory’ and ‘ActiveHistory’ cache statistics that take too long to calculate otherwise; ‘EventPassword’ contains authentication data for tagging. Diagram produced with MySql Workbench. [204]

Appendix F

TIGRESS TRANSFORMS

Transform	Description
Virtualize	Translates a function into virtual machine code that is interpreted at runtime.
Jit	Translates a function into virtual machine code that is compiled to machine code at runtime.
Dynamic Jit	Translates a function into binary code which is decoded and re-encoded at runtime.
Flatten	Removes direct control flow from a function.
Merge	Merges two functions into a single function.
Split	Splits a function into two functions.
AddOpaque	Adds bogus control flow to a function.
EncodeData	Encodes variables so that their values are hidden at runtime.
Anti Branch Analysis	Impedes branch analysis using, for example, <i>branch functions</i> [214].
Anti Alias Analysis	Impedes alias analysis by adding indirect references.
Anti Taint Analysis	Impedes dynamic taint analysis by copying variables using, for example, implicit flow [298].

Table F.1: *Tigress* Transforms, reproduced from [307].

Appendix G

REVENGE STUDY CHALLENGE GENERATION SCRIPT 1: EASY PASSWORD EXTRACTION

```
tigress \
    —Verbosity=1 \
    —Environment=x86_64:Linux:Gcc:4.6 \
    —Transform=RandomFuns \
        —RandomFunsName=SECRET \
        —RandomFunsType=long \
        —RandomFunsInputSize=2 \
        —RandomFunsOutputSize=1 \
        —RandomFunsCodeSize=25 \
        —RandomFunsLocalStorageSize=2 \
        —RandomFunsLocalDynamicStateSize=2 \
        —RandomFunsGlobalStaticStateSize=2 \
        —RandomFunsGlobalDynamicStateSize=2 \
        —RandomFunsLoopSize=20 \
        —RandomFunsActivationCodeCheckCount=1 \
        —RandomFunsActivationCode=224720257 \
        —RandomFunsFailureKind=segv \
    —out=medium.c empty.c

tigress \
    —Verbosity=1 \
    —Environment=x86_64:Linux:Gcc:4.6 \
    —FilePrefix=virtswentimp \
    —Transform=InitEntropy \
        —Functions=main \
    —Transform=InitEntropy \
        —Functions=main \
    —Transform=InitImplicitFlow \
        —Functions=main \
        —InitImplicitFlowKinds= \

```

```

trivial_thread_1 , trivial_counter \
—Transform=Virtualize \
—Functions=SECRET_1 \
—VirtualizeDispatch=switch \
—VirtualizeShortIdent=true \
—VirtualizeOperands=stack \
—VirtualizeMaxDuplicateOps=0 \
—VirtualizeSuperOpsRatio=0.0 \
—VirtualizeMaxMergeLength=0 \
—VirtualizeNumberOfBogusFuns=0 \
—VirtualizeBogusLoopKinds=* \
—VirtualizePerformance=IndexedStack \
—VirtualizeStackSize=100 \
—VirtualizeBogusLoopIterations=0 \
—VirtualizeImplicitFlow= \
"(compose (single counter_signal) \
(single bitcopy_loop) \
(single counter_float))" \
—Transform=InitOpaque \
—Functions=main \
—InitOpaqueCount=2 \
—InitOpaqueStructs=list ,array ,env \
—Transform=UpdateOpaque \
—Functions=SECRET_1 \
—UpdateOpaqueCount=10 \
—Transform=AddOpaque \
—Functions=SECRET_1 \
—AddOpaqueKinds=call ,bug ,true ,junk ,question \
—AddOpaqueCount=25 \
—Transform=CleanUp \
—CleanUpKinds=annotations \
medium.c —out=crack.c

```

clang -5.0 -lm -lpthread -static -o crack.exe crack.c

strip -s crack.exe

Appendix H

REVENGE STUDY CHALLENGE GENERATION SCRIPT 2: MEDIUM PASSWORD EXTRACTION

```
tigress \
    —Verbosity=1 \
    —Environment=x86_64:Linux:Gcc:4.6 \
    —Transform=RandomFuns \
        —RandomFunsName=SECRET \
        —RandomFunsType=long \
        —RandomFunsInputSize=2 \
        —RandomFunsOutputSize=1 \
        —RandomFunsCodeSize=25 \
        —RandomFunsLocalStorageSize=2 \
        —RandomFunsLocalDynamicStateSize=2 \
        —RandomFunsGlobalStaticStateSize=2 \
        —RandomFunsGlobalDynamicStateSize=2 \
        —RandomFunsLoopSize=20 \
        —RandomFunsActivationCodeCheckCount=1 \
        —RandomFunsActivationCode=3089123980 \
        —RandomFunsFailureKind=segv \
        —out=medium.c empty.c

tigress \
    —Verbosity=1 \
    —Environment=x86_64:Linux:Gcc:4.6 \
    —FilePrefix=virtswentimp \
    —Transform=InitEntropy \
        —Functions=main \
    —Transform=InitEntropy \
        —Functions=main \
    —Transform=InitImplicitFlow \
        —Functions=main \
        —InitImplicitFlowKinds=
```

```

trivial_thread_1 , trivial_counter \
—Transform=Virtualize \
—Functions=SECRET_1 \
—VirtualizeDispatch=switch \
—VirtualizeShortIdent=true \
—VirtualizeOperands=stack \
—VirtualizeMaxDuplicateOps=0 \
—VirtualizeSuperOpsRatio=0.0 \
—VirtualizeMaxMergeLength=0 \
—VirtualizeNumberOfBogusFuns=0 \
—VirtualizeBogusLoopKinds=* \
—VirtualizePerformance=IndexedStack \
—VirtualizeStackSize=100 \
—VirtualizeBogusLoopIterations=0 \
—VirtualizeImplicitFlow= \
"(compose (single counter_signal) \
(single bitcopy_loop) \
(single counter_float))" \
—Transform=InitOpaque \
—Functions=main \
—InitOpaqueCount=2 \
—InitOpaqueStructs=list ,array ,env \
—Transform=UpdateOpaque \
—Functions=SECRET_1 \
—UpdateOpaqueCount=10 \
—Transform=AddOpaque \
—Functions=SECRET_1 \
—AddOpaqueKinds=call ,bug ,true ,junk ,question \
—AddOpaqueCount=25 \
—Transform=SelfModify \
—Skip=false \
—Functions=SECRET_1 \
—SelfModifySubExpressions=false \
—SelfModifyBogusInstructions=10 \
—Transform=CleanUp \
—CleanUpKinds=annotations \
medium.c —out=crack.c

```

```
clang-5.0 -lm -lpthread -static -Wl,--omagic -o crack.exe crack.c  
strip -s crack.exe
```

Appendix I

REVENGE STUDY CHALLENGE GENERATION SCRIPT 3: DEOBFUSCATION AND DECOMPILATION

```
tigress \
    —FilePrefix=x \
    —Verbosity=1 \
    —Seed=393758450 \
    —Environment=x86_64:Linux:Gcc:4.6 \
    —Verbosity=1 \
    —Transform=RandomFuns \
        —RandomFunsName=SECRET \
        —RandomFunsType=double \
        —RandomFunsInputSize=1 \
        —RandomFunsOutputSize=1 \
        —RandomFunsFailureKind=segv \
        —RandomFunsCodeSize=50 \
        —RandomFunsLocalStaticStateSize=4 \
        —RandomFunsLocalDynamicStateSize=4 \
        —RandomFunsGlobalStaticStateSize=4 \
        —RandomFunsGlobalDynamicStateSize=4 \
        —RandomFunsLoopSize=20 \
    —out=large.c empty.c

tigress \
    —FilePrefix=y \
    —Seed=92120384355 \
    —Statistics=0 \
    —Verbosity=0 \
    —Environment=x86_64:Linux:Gcc:4.6 \
    —Verbosity=3 \
    —Transform=InitEntropy \
    —Functions=main \
    —InitEntropyKinds=vars \
```

```
—Transform=InitOpaque \
—Functions=main \
—InitOpaqueStructs=list ,array ,env      \
—Transform=Virtualize \
    —Functions=SECRET_1 \
    —VirtualizeDispatch=indirect \
    —VirtualizeShortIdent=true \
    —VirtualizeOperands=mixed \
    —VirtualizeMaxDuplicateOps=10 \
    —VirtualizeSuperOpsRatio=2.0 \
    —VirtualizeMaxMergeLength=10 \
    —VirtualizeNumberOfBogusFuns=0 \
    —VirtualizeBogusLoopKinds=* \
    —VirtualizePerformance=IndexedStack \
    —VirtualizeStackSize=100 \
    —VirtualizeBogusLoopIterations=0 \
large.c —out=deobfuscate.c
```

```
gcc -o deobfuscate.exe deobfuscate.c
```

```
strip -s deobfuscate.exe
```

Appendix J

REVEnGE CTF STUDY DECOMPILATION CHALLENGE CONTROL FLOW GRAPHS

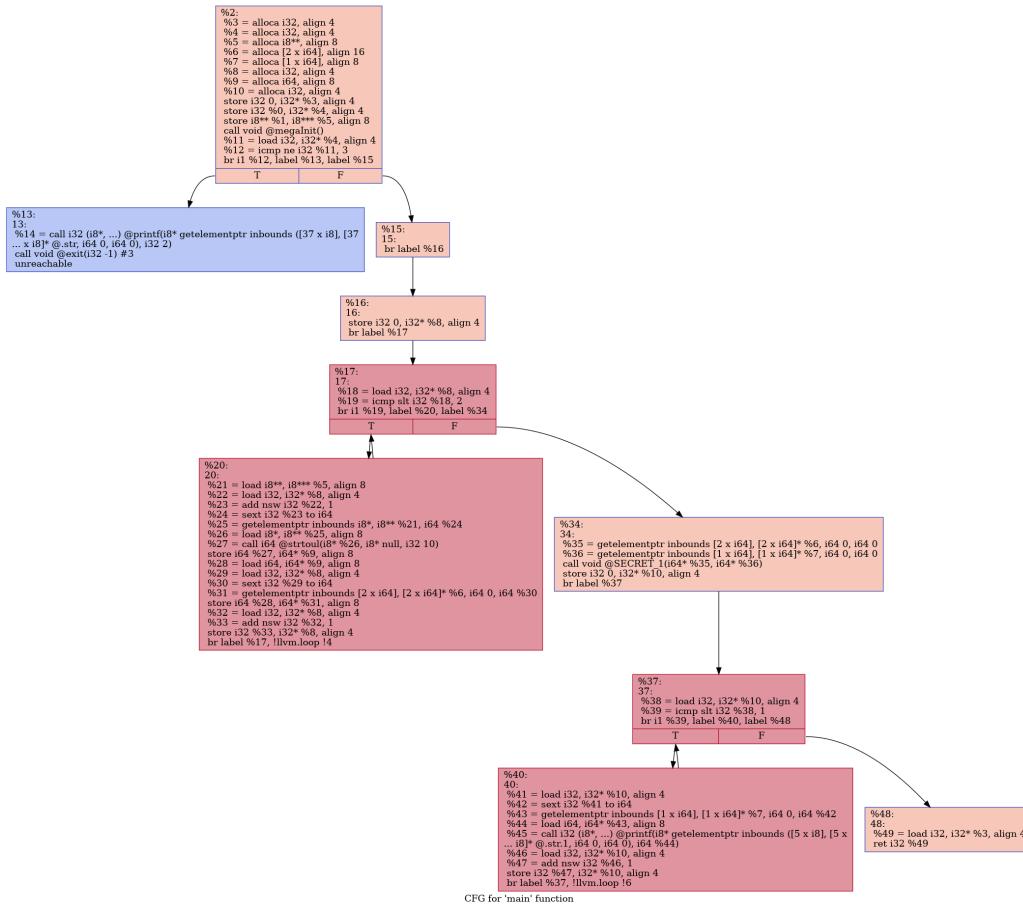


Figure J.1: The non-obfuscated program generated by the script in Appendix I and solved during the RevEngE CTF study. The code generating this program is about 20 KB in size.

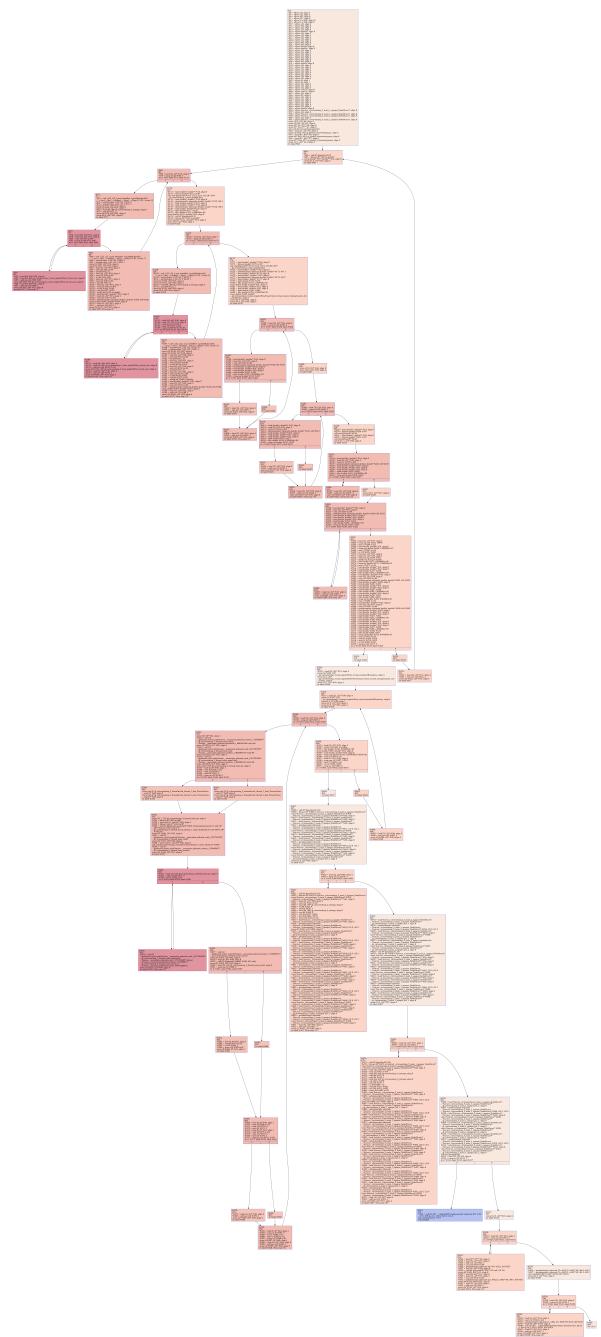


Figure J.2: The obfuscated version of the program generated by the script in Appendix I and solved during the RevEngE CTF study. The code generating this program is about 461 KB in size.

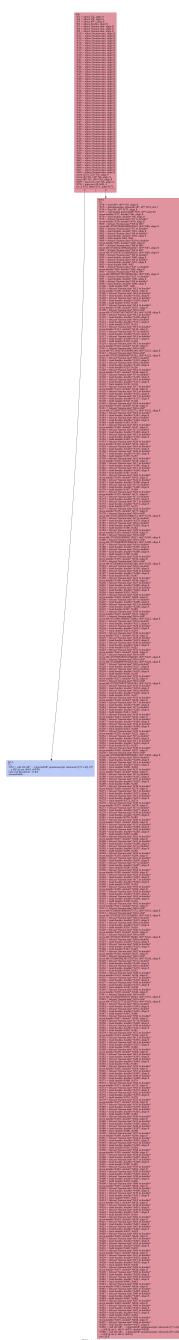


Figure J.3: The participant's successful submission for the deobfuscation/decompilation challenge generated by the script in Appendix I. The code generating this program is about 4 KB in size.

Appendix K

GHENT UNIVERSITY REVERSE ENGINEERING ASSIGNMENT

3/9/22, 1:54 PM

Final Reverse Engineering Lab - E017941A - Software Hacking and Protection



E017941A - Software Hacking and Protection



TZ



Final Reverse Engineering Lab

Instructions

REPORT AND GRADING

You need to file a text response. For each of the two challenges, report either a valid key or, in case you didn't find a valid key, you write down (at a higher level of abstraction) the rules/requirements that you already learned about valid keys, i.e., what you already learned by reverse engineering.

You receive full grades for any meaningful (not copied) answer.

INDIVIDUAL WORK

Even though you are assigned to a group, you need to participate on your own, i.e., without communicating with your colleagues about the assignment. The group assignment is only in place to partition the differently obfuscated binaries among all of you in a random, but evenly distributed manner.

TEACHING ASSISTANCE AND TECHNICAL ASSISTANCE

In this remote lab, no teaching assistants are available to help you with the content of the assignment, i.e., with the reverse engineering tasks themselves. You are on your own in this lab, which revisits concepts you already obtained experience on in previous labs.

Only for technical issues (such as problems downloading binaries or installing our software), you can contact the teaching assistant, Bert Abrath. Please do so in a chat on MS Teams. Bert will be available to answer questions in the chat or in subsequent calls throughout the whole afternoon.

ASSIGNMENT AND RESOURCES

First of all, if you want to participate to the experiment, please make sure you also have a look at the experiment participation instructions.

3/9/22, 1:54 PM

Final Reverse Engineering Lab - E017941A - Software Hacking and Protection

In this lab, we want you to attack two binaries. The grading is purely based on participation.

Please attack these binaries in the same VM you have been using for previous reverse engineering and obfuscation labs.

Do not use any brute-forcing attacks! (The input spaces are too large anyway.)

At the start of the lab, download the binaries here. Unpack the download to obtain the individual binaries and libraries they depend on. (Update: Please note that at the start of the lab session, something was wrong with the provided link, so if you downloaded an older assignment, please reload. We're sorry ...)

Challenge 1:

- For the first challenge, all of you target the same binary named challenge_1.
- The program takes two command-line arguments. They represent keys, which are checked independently and separately against some rules. The two checks correspond to two functions that are called consecutively.
- The first check does not need to be reverse engineered. Its key correctness rules are as follows:
 - Key format: X_Y where X and Y are two decimal integers.
 - X and Y form a solution to the following equation:

$$2X^3Y^3 + 400X^3Y^2 + 37X^2Y^3 - 10081X^2Y^2 + 2XY + 400X + 37Y - 10081 = 0.$$
 - An example of a valid key is -8759_-201. You can use this key to execute the program with a successful first check.
 - You could easily have found other keys using an online tool such as Wolfram Alpha.
- The second check is your reverse engineering task: Try to find a valid input. All you know upfront is that the format of the second key is ABCD-EFGH-IJKL-MNOP, where all characters stand for one digit 0-9. One valid key is 0001-6347-0001-0251. You obviously need to find another one, and we mean a significantly different one that is not, e.g., a simple permutation of the above key's parts.
- You remember some of the reverse engineering tools you have used previously on (obfuscated) software reverse engineering challenges, such as

3/9/22, 1:54 PM

Final Reverse Engineering Lab - E017941A - Software Hacking and Protection

IDA Pro and GDB? Well, perhaps those will be handy here as well ;-)

Challenge 2:

- For the second challenge, we have foreseen three slightly different challenges. In overall complexity they are the same, and they still consist of command-line programs that perform key checks on two command-line arguments. The rules that the two keys need to meet, are independent of each other. In other words, the correctness and other attributes of one key do not influence whether the other key is correct. The implementations of the two checks are not necessarily independent, however.
- **The challenge you need to work on (challenge_2a, challenge_2b, or challenge_2c) corresponds to your group for this lab.** You can find it on Ufora in the *Final lab challenge distribution* group category.
- Again, you don't need to reverse engineer the first check, you only need to reverse engineer the check on the second argument, trying to find a valid input for it. A valid existing input of the check on the second argument is 76133-169499-262865-356231-449597. Again, you need to find another one, and we mean a significantly different one that is not, e.g., a simple permutation of the above key's parts.
- As for the first check that you don't need to reverse engineer, we provide the following valid keys, which differ from one challenge to another. You can use these keys to execute the binaries without having them fail in the check on the first argument.
 - challenge 2a:
 - key 1: bart-barts-bert-bar-applets-berts-apple-bars-thomas-tiger-tigersss-rabbit
 - key 2: bart-barto-bongo-bar-applets-berts-apple-bars-thoma-tiger-tigersss-stevadore
 - key 3: bar-bari-yule-bart-applets-berts-apple-bars-endtime-tigers-tigersss-fresh
 - challenge 2b:
 - key 1: 018-028-038-048-058_04-102400
 - key 2: 417-427-437-407-457_45-2100875
 - key 3: 111-111-111-111-111_15-100000
 - challenge 2c:

3/9/22, 1:54 PM

Final Reverse Engineering Lab - E017941A - Software Hacking and Protection

- key 1: bart-barts-bert-bar-applets-berts-apple-bars-thomas-tiger-tigerss-rabbit
- key 2: bart-barto-bongo-bar-applets-berts-apple-bars-thomas-tiger-tigerss-stevadore
- key 3: bar-bari-yule-bart-applets-berts-apple-bars-endtime-tigers-tigerss-fresh
- Obviously, for this challenge it might be good to consider the same types of tools as could have been useful for reversing the key checks of the first challenge and that you used before in this course (IDA Pro, gdb, Wolfram Alpha, strings, objdump, ltrace, ...).

► Add Attachments

 Reflect in ePortfolio



Activity Details	Learning Objectives	Completion Summary
<input checked="" type="checkbox"/> Visibility No Completion Tracking ▾ ⌚ Due 15 December 2021 5:30 PM Options Unlimited files All submissions are kept Reflecting in ePortfolio is enabled		Assessment Labs: Final Reverse Engineering Lab - / 10  Participation Lab

Last Modified 13/12/2021 1:40 PM

Appendix L

GHENT UNIVERSITY ASSIGNMENT RESULTS

User	Session	Time	Tasks	Windows	Processes	Screenshots	Mouse	Key
1002	...a7cfb96	36	0	2	4921	36	342	719
1002	...c3b43c70	1	0	2	196	3	5	108
1002	...a9902ea0	0	0	2	190	2	3	0
1002	...3908c6f7	0	0	2	199	2	4	0
1003	...9ab918f6	4	0	2	2522	18	75	129
12345	...cf9e0b61	14	2	11	10429	58	99	1117
12345	...fa7afe9c	7	0	11	5446	30	60	277
12345	...1eb72dfc	63	1	64	46640	265	527	2152
12345	...1f82430c	7	1	11	5871	33	129	533
12345	...1684cb38	11	1	37	7956	50	290	872
12345	...0ce33286	5	0	50	5711	40	78	329
12345	...09539f6f	4	0	99	3034	49	88	135
12345	...574ad300	1	0	6	700	8	12	297
12345	...90aa6a65	3	0	4	1519	13	65	144
12345	...2e757b96	1	0	2	717	7	38	300
12345	...0c901383	7	0	2	3806	31	92	372
12345	...e3d33665	5	1	2	2868	22	46	672
12345	...264484a6	10	0	17	4880	44	176	663
12345	...e22f5dcd	52	0	11	32546	213	23	801
12345	...d24e23af	120	0	20	19207	133	42	462
12345	...1d86db24	116	0	80	73839	494	184	966

12345	... b7c82767	1	0	2	723	7	20	369
12345	... 1bcecd3b	28	2	108	14981	132	206	450
12345	... fe44f649	36	0	2	17433	145	8	87
12345	... 29db10e8	5	0	5	3130	22	4	1362
12345	... a3c5f75d	9	0	22	6169	47	76	153
1169420789693	... c5811ac5	197	1	2	96123	780	3745	10002
1151659450775	... 95256761	102	11	77	48451	420	1818	15233
1151659450775	... 44723ab0	117	10	13	54808	469	1774	14395
1216502329770	... e97ab3ea	265	19	2	122666	1050	2621	14768
1109548117664	... ba824e2a	264	3	15	137250	1059	1726	18207
1056106216780	... e29f65a9	29	4	8	661	121	0	0
1149671050952	... 0561eb04	40	1	2	17917	161	44	3537
1041477566779	... fb22e1bc	29	1	2	14671	119	234	1707
1041477566779	... 9b79d692	6	0	2	3336	27	146	657
1041477566779	... 82e2b3e9	221	4	2	101404	804	297	17820
1050474970484	... aef2d260	266	6	53	131725	1133	3237	11965
1145799096526	... 379f92b5	197	0	14	93921	790	1121	7927
1181313745033	... a9cc3d15	225	78	2	96265	838	3942	20622
1140210052348	... 77a8d302	170	2	2	80861	673	3341	14936
1233929119994	... 767d66ee	239	8	15	83622	701	2164	17055
1171952646901	... 0c3a92a3	178	0	2	92981	712	3901	22664
1042918769867	... 8dcbb20df	4	0	35	1985	25	14	411
1042918769867	... 095cae1c	2	0	3	821	8	8	156
1042918769867	... d8795b1d	2	0	2	1144	10	38	131
1042918769867	... 169d14da	120	4	61	56076	493	3184	13871
1042918769867	... 3ffe673a	7	0	9	1322	13	61	748
1042918769867	... dddbb25b3	57	2	68	27798	243	1908	8861
1042918769867	... 7469ff55	3	0	2	1620	15	18	510

1042918769867	...c2a7f580	99	6	168	36402	330	667	16507
1052233862958	...dce1f815	126	0	37	57490	468	3922	10507
1037206761638	...202ed7f8	102	4	15	47470	411	1145	2249
1037206761638	...a794f7aa	3	0	11	1470	15	72	278
1037206761638	...2a20f02e	0	0	2	195	2	4	0
1037206761638	...46be9769	87	6	0	3855	349	0	0
1037206761638	...f4d59ee1	4	0	2	1773	16	60	0
1184710457397	...f4a80bd8	0	0	3	164	3	2	0
1184710457397	...7a6a6f8d	251	36	3	116901	1002	2984	14240
1187264017243	...8d4c45ef	231	16	32	110170	929	2283	16396
1155601377584	...1bb5dbfe	92	0	34	43259	377	1104	11039
1036235334128	...0f25770f	8	0	8	4367	33	80	825
1228407680657	...884472e1	249	0	2	113486	978	2652	6641
1191061470042	...ae3cb7f8	62	1	20	15938	145	706	2943
1191061470042	...ba40a5a1	17	0	2	7996	70	278	1045
... Total		4617	231	1308	2103997	17696	53993	312322

Table L.1: A summary view of subjects' activity from the Ghent University Assignment. Time is measured in minutes, while other columns are data entry counts.

BIBLIOGRAPHY

- [1] MA Abakumov and Pavel Mikhailovich Dovgalyuk. Stealth debugging of programs in qemu emulator with windbg debugger. *Труды Института системного программирования РАН*, 30(3), 2018.
- [2] Yasemin Acar, Christian Stransky, Dominik Wermke, Charles Weir, Michelle L Mazurek, and Sascha Fahl. Developers need support, too: A survey of security advice for software developers. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 22–26. IEEE, 2017.
- [3] Ehsan Aghaei and Gursel Serpen. Host-based anomaly detection using eigen-traces feature extraction and one-class classification on system call trace data. *arXiv preprint arXiv:1911.11284*, 2019.
- [4] Dan Aharoni. What you see is what you get the influence of visualization on the perception of data structures. *DOCUMENT RESUME*, 11(4):10, 2000.
- [5] Mohsen Ahmadvand, Alexander Pretschner, and Florian Kelbert. A taxonomy of software integrity protection techniques. In *Advances in Computers*, volume 112, pages 413–486. Elsevier, 2019.
- [6] Nesreen K Ahmed, Amir F Atiya, Neamat El Gayar, and Hisham El-Shishiny. An empirical comparison of machine learning models for time series forecasting. *Econometric Reviews*, 29(5-6):594–621, 2010.
- [7] Adnan Akhunzada, Mehdi Sookhak, Nor Badrul Anuar, Abdullah Gani, Ejaz Ahmed, Muhammad Shiraz, Steven Furnell, Amir Hayat, and Muhammad Khurram Khan. Man-at-the-end attacks: Analysis, taxonomy, human aspects, motivation and future directions. *Journal of Network and Computer Applications*, 48:44–57, 2015.
- [8] Hamad Al-Mohannadi, Qublai Mirza, Anitta Namanya, Irfan Awan, Andrea Cullen, and Jules Disso. Cyber-attack modeling analysis techniques: An overview. In *2016 IEEE 4th international conference on future internet of things and cloud workshops (FiCloudW)*, pages 69–76. IEEE, 2016.
- [9] Hasan Al-Qadeeb, Q Abu Al-Haija, and Noor A Jebril. Software simulation of variable size message encryption based rsa crypto-algorithm using ms. c#

- .net. *International Journal of Cyber-Security and Digital Forensics(IJCSDF)*, DOI: <http://dx.doi.org/10.17781P>, 2241, 2017.
- [10] Gorjan Alagic, Zvika Brakerski, Yfke Dulek, and Christian Schaffner. Impossibility of quantum virtual black-box obfuscation of classical circuits. In *Annual International Cryptology Conference*, pages 497–525. Springer, 2021.
 - [11] Jean-Baptiste Alayrac, Piotr Bojanowski, Nishant Agrawal, Josef Sivic, Ivan Laptev, and Simon Lacoste-Julien. Unsupervised learning from narrated instruction videos. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4575–4583, 2016.
 - [12] Laila Almutairi, Liang Hong, and Sachin Shetty. Security analysis of multiple sdn controllers based on stochastic petri nets. In *2019 Spring Simulation Conference (SpringSim)*, pages 1–12. IEEE, 2019.
 - [13] M Alnuem, J Mellor, and R Fretwell. New algorithm to control tcp behavior over lossy links. In *2009 International Conference on Advanced Computer Control*, pages 236–240. IEEE, 2009.
 - [14] Mashael AlSabah and Ian Goldberg. Performance and security improvements for tor: A survey. *ACM Computing Surveys (CSUR)*, 49(2):1–36, 2016.
 - [15] Aubrey Alston. Concolic execution as a general method of determining local malware signatures. *arXiv preprint arXiv:1705.05514*, 2017.
 - [16] Bertrand Anckaert, Mariusz Jakubowski, and Ramarathnam Venkatesan. Proteus: virtualization for diversified tamper-resistance. In *Proceedings of the ACM workshop on Digital rights management*, pages 47–58, 2006.
 - [17] Bertrand Anckaert, Matias Madou, Bjorn De Sutter, Bruno De Bus, Koen De Bosschere, and Bart Preneel. Program obfuscation: a quantitative approach. In *Proceedings of the 2007 ACM workshop on Quality of protection*, pages 15–20, 2007.
 - [18] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 583–600, 2016.

- [19] Ludovic Apvrille and Yves Roudier. Sysml-sec attack graphs: compact representations for complex attacks. In *International Workshop on Graphical Models for Security*, pages 35–49. Springer, 2015.
- [20] Oluseun Omotola Aremu, David Hyland-Wood, and Peter Ross McAree. A machine learning approach to circumventing the curse of dimensionality in discontinuous time series machine data. *Reliability Engineering & System Safety*, 195:106706, 2020.
- [21] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. Posix abstractions in modern operating systems: The old, the new, and the missing. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–17, 2016.
- [22] Maxime Audinot, Sophie Pinchinat, and Barbara Kordy. Is my attack tree correct? In *European Symposium on Research in Computer Security*, pages 83–102. Springer, 2017.
- [23] Pooneh Nikkhah Bahrami, Ali Dehghantanha, Tooska Dargahi, Reza M Parizi, Kim-Kwang Raymond Choo, and Hamid HS Javadi. Cyber kill chain-based taxonomy of advanced persistent threat actors: analogy of tactics, techniques, and procedures. *Journal of information processing systems*, 15(4):865–889, 2019.
- [24] Daniel V Bailey, Lejla Batina, Daniel J Bernstein, Peter Birkner, Joppe W Bos, Hsieh-Chung Chen, Chen-Mou Cheng, Gauthier Van Damme, Giacomo de Meulenaer, Luis J Dominguez Perez, et al. Breaking ecc2k-130. *IACR Cryptol. ePrint Arch.*, 2009:541, 2009.
- [25] Ahmet BALCI, Dan UNGUREANU, and Jaromír VONDRUŠKA. *Malware Reverse Engineering Handbook*. CCDCOE, 2020.
- [26] Sebastian Banescu, Martín Ochoa, and Alexander Pretschner. A framework for measuring software obfuscation resilience against automated attacks. In *Software Protection (SPRO), 2015 IEEE/ACM 1st International Workshop on*, pages 45–51. IEEE, 2015.
- [27] Lingfeng Bao, Jing Li, Zhenchang Xing, Xinyu Wang, and Bo Zhou. Reverse engineering time-series interaction data from screen-captured videos. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 399–408. IEEE, 2015.

- [28] Lingfeng Bao, Zhenchang Xing, Xin Xia, David Lo, Minghui Wu, and Xiaohu Yang. psc2code: Denoising code extraction from programming screen-casts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(3):1–38, 2020.
- [29] Boaz Barak. Hopes, fears, and software obfuscation. *Communications of the ACM*, 59(3):88–96, 2016.
- [30] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im) possibility of obfuscating programs. In *Annual international cryptology conference*, pages 1–18. Springer, 2001.
- [31] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im) possibility of obfuscating programs. *Journal of the ACM (JACM)*, 59(2):1–48, 2012.
- [32] Titus Barik, Brittany Johnson, and Emerson Murphy-Hill. I heart hacker news: expanding qualitative research findings by analyzing social news websites. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 882–885, 2015.
- [33] Sean Barnum. Standardizing cyber threat intelligence information with the structured threat information expression (stix). *Mitre Corporation*, 11:1–22, 2012.
- [34] C Basile et al. D5. 11 aspire framework report. *POLITO,” techreport*, Nov, 2016.
- [35] Cataldo Basile, Daniele Cavanese, Leonardo Regano, Paolo Falcarin, and Bjorn De Sutter. A meta-model for software protections and reverse engineering attacks. *Journal of Systems and Software*, 2019.
- [36] Leonid Batyuk, Markus Herpich, Seyit Ahmet Camtepe, Karsten Raddatz, Aubrey-Derrick Schmidt, and Sahin Albayrak. Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications. In *2011 6th International Conference on Malicious and Unwanted Software*, pages 66–72. IEEE, 2011.
- [37] Erick Bauman, Zhiqiang Lin, Kevin W Hamlen, et al. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *NDSS*, 2018.

- [38] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1):67–77, 2006.
- [39] Rachel Bellamy, Bonnie John, John Richards, and John Thomas. Using cogtool to model programming tasks. In *Evaluation and Usability of Programming Languages and Tools*, pages 1–6. ACM Onward, 2010.
- [40] Jürgen Bernard, Eduard Dobermann, Markus Bögl, Martin Röhlig, Anna Vögele, and Jörn Kohlhammer. Visual-interactive segmentation of multivariate time series. In *EuroVA@ EuroVis*, pages 31–35, 2016.
- [41] Daniel J Bernstein, Hsieh-Chung Chen, Chen-Mou Cheng, Tanja Lange, Ruben Niederhagen, Peter Schwabe, and Bo-Yin Yang. Ecc2k-130 on nvidia gpus. In *International conference on cryptology in India*, pages 328–346. Springer, 2010.
- [42] Eike Best and Raymond Devillers. Sequential and concurrent behaviour in petri net theory. *Theoretical Computer Science*, 55(1):87–136, 1987.
- [43] Eike Best, Raymond Devillers, and Maciej Koutny. *Petri net algebra*. Springer Science & Business Media, 2013.
- [44] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. Orbs: Language-independent program slicing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 109–120, 2014.
- [45] Fabrizio Biondi, Sébastien Josse, Axel Legay, and Thomas Sirvent. Effectiveness of synthesis in concolic deobfuscation. *Computers & Security*, 70:500–515, 2017.
- [46] Henk Birkholz, Stefan Edelkamp, Florian Junge, and Karsten Sohr. Efficient automated generation of attack trees from vulnerability databases. In *Working Notes for the 2010 AAAI Workshop on Intelligent Security (SecArt)*, pages 47–55, 2010.
- [47] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. Power struggles: Revisiting the risc vs. cisc debate on contemporary arm and x86 architectures. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12. IEEE, 2013.

- [48] Estuardo Alpirez Bock, Alessandro Amadori, Chris Brzuska, and Wil Michiels. On the security goals of white-box cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 327–357, 2020.
- [49] Alexandru Boicea, Florin Radulescu, and Laura Ioana Agapin. Mongodb vs oracle–database comparison. In *2012 third international conference on emerging intelligent data and web technologies*, pages 330–335. IEEE, 2012.
- [50] Dan Boneh et al. Twenty years of attacks on the rsa cryptosystem. *Notices of the AMS*, 46(2):203–213, 1999.
- [51] Gianluca Bontempi, Souhaib Ben Taieb, and Yann-Aël Le Borgne. Machine learning strategies for time series forecasting. In *European business intelligence summer school*, pages 62–77. Springer, 2012.
- [52] Fabrice Boudot, Pierrick Gaudry, Aurore Guillevic, Nadia Heninger, Emmanuel Thomé, and Paul Zimmermann. Comparing the difficulty of factorization and discrete logarithm: a 240-digit experiment. In *Annual International Cryptology Conference*, pages 62–91. Springer, 2020.
- [53] Zvika Brakerski and Guy N Rothblum. Virtual black-box obfuscation for all circuits via generic graded encoding. In *Theory of Cryptography Conference*, pages 1–25. Springer, 2014.
- [54] Rodrigo Rubira Branco, Gabriel Negreira Barbosa, and Pedro Drimel Neto. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. *Black Hat*, 1:1–27, 2012.
- [55] Wyseur Brecht. White-box cryptography: hiding keys in software. *NAGRA Kudelski Group*, 2012.
- [56] Kirk Bresniker, Ada Gavrilovska, James Holt, Dejan Milojicic, and Trung Tran. Grand challenge: Applying artificial intelligence and machine learning to cybersecurity. *Computer*, 52(12):45–52, 2019.
- [57] Hennie Brugman, Albert Russel, and Xd Nijmegen. Annotating multi-media/multi-modal resources with elan. In *LREC*, pages 2065–2068, 2004.
- [58] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. Bap: A binary analysis platform. In *International Conference on Computer Aided Verification*, pages 463–469. Springer, 2011.

- [59] Adam Bryant. Toward detecting novel software attacks by using constructs from human cognition. In *Proceedings of the 3rd International Conference on Information Warfare and Security, Peter Kiewit Institute, University of Nebraska, Omaha USA*, pages 24–25, 2008.
- [60] Adam R Bryant. *Understanding how reverse engineers make sense of programs from assembly language representations*. Air Force Institute of Technology, 2012.
- [61] Johannes Buchmann, Niklas Büscher, Florian Göpfert, Stefan Katzenbeisser, Juliane Krämer, Daniele Micciancio, Sander Siim, Christine van Vredendaal, and Michael Walter. Creating cryptographic challenges using multi-party computation: The lwe challenge. In *Proceedings of the 3rd ACM International Workshop on ASIA Public-Key Cryptography*, pages 11–20, 2016.
- [62] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.
- [63] Cristian Cadar and Martin Nowack. Klee symbolic execution engine in 2019. *International Journal on Software Tools for Technology Transfer*, pages 1–4, 2020.
- [64] Carlos Caldera et al. *Towards an automated attack tree generator for the IoT*. PhD thesis, Massachusetts Institute of Technology, 2017.
- [65] Sergio Caltagirone, Andrew Pendergast, and Christopher Betz. The diamond model of intrusion analysis. Technical report, Center For Cyber Intelligence Analysis and Threat Research Hanover Md, 2013.
- [66] Daniele Canavese, Leonardo Regano, Cataldo Basile, Bart Coppens, and Bjorn De Sutter. Software protection as a risk analysis process. *arXiv preprint arXiv:2011.07269*, 2020.
- [67] Ran Canetti. Towards realizing random oracles: Hash functions that hide all partial information. In *Annual International Cryptology Conference*, pages 455–469. Springer, 1997.
- [68] Gerardo Canfora and Massimiliano Di Penta. New frontiers of reverse engineering. In *Future of Software Engineering (FOSE'07)*, pages 326–341. IEEE, 2007.

- [69] Chris Cannam, Christian Landone, and Mark Sandler. Sonic visualiser: An open source application for viewing, analysing, and annotating music audio files. In *Proceedings of the 18th ACM international conference on Multimedia*, pages 1467–1468, 2010.
- [70] David Carasso. Exploring splunk. *published by CITO Research, New York, USA, ISBN* 978–0, 2012.
- [71] James A Carlson, Arthur Jaffe, and Andrew Wiles. *The millennium prize problems*. Citeseer, 2006.
- [72] Gustavo Carneiro, Antoni B Chan, Pedro J Moreno, and Nuno Vasconcelos. Supervised learning of semantic classes for image annotation and retrieval. *IEEE transactions on pattern analysis and machine intelligence*, 29(3):394–410, 2007.
- [73] Brian Caswell and Jay Beale. *Snort 2.1 intrusion detection*. Elsevier, 2004.
- [74] Stefania Cavallar, Bruce Dodson, Arjen K Lenstra, Walter Lioen, Peter L Montgomery, Brian Murphy, Herman Te Riele, Karen Aardal, Jeff Gilchrist, Gérard Guillerm, et al. Factorization of a 512-bit rsa modulus. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–18. Springer, 2000.
- [75] Mariano Ceccato. On the need for more human studies to assess software protection. In *Workshop on Continuously Upgradeable Software Security and Protection*, pages 55–56, 2014.
- [76] Mariano Ceccato, Massimiliano Di Penta, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques. *Empirical Software Engineering*, 19(4):1040–1074, 2014.
- [77] Mariano Ceccato, Massimiliano Di Penta, Jasvir Nagra, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. The effectiveness of source code obfuscation: An experimental assessment. In *Program Comprehension, 2009. ICPC’09. IEEE 17th International Conference on*, pages 178–187. IEEE, 2009.
- [78] Mariano Ceccato and Riccardo Scandariato. Static analysis and penetration testing from the perspective of maintenance teams. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–6, 2016.

- [79] Mariano Ceccato, Paolo Tonella, Cataldo Basile, Bart Coppens, Bjorn De Sutter, Paolo Falcarin, and Marco Torchiano. How professional hackers understand protected code while performing attack tasks. In *Proceedings of the 25th International Conference on Program Comprehension*, pages 154–164. IEEE Press, 2017.
- [80] Mariano Ceccato, Paolo Tonella, Cataldo Basile, Paolo Falcarin, Marco Torchiano, Bart Coppens, and Bjorn De Sutter. Understanding the behaviour of hackers while performing attack tasks in a professional setting and in a public challenge. *Empirical Software Engineering*, 24(1):240–286, 2019.
- [81] Nilotpal Chakraborty and Ezhil Kalaimannan. Minimum cost security measurements for attack tree based threat models in smart grid. In *2017 IEEE 8th Annual Ubiquitous Computing, Electronics and Mobile Communication Conference (UEMCON)*, pages 614–618. IEEE, 2017.
- [82] Rahul Chandran and Wei Q Yan. Attack graph analysis for network anti-forensics. *International Journal of Digital Crime and Forensics (IJDCF)*, 6(1):28–50, 2014.
- [83] Chih-Fan Chen, Theofilos Petsios, Marios Pomonis, and Adrian Tang. Confuse: Llvm-based code obfuscation, 2013.
- [84] Gengbiao Chen, Zhengwei Qi, Shiqiu Huang, Kangqi Ni, Yudi Zheng, Walter Binder, and Haibing Guan. A refined decompiler to generate c code with high readability. *Software: Practice and Experience*, 43(11):1337–1358, 2013.
- [85] Jieshan Chen, Mulong Xie, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, and Guoqiang Li. Object detection for graphical user interface: old fashioned or deep learning or a combination? In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1202–1214, 2020.
- [86] Liang Chen, Yuyao Zhai, Qiuyan He, Weinan Wang, and Minghua Deng. Integrating deep supervised, self-supervised and unsupervised learning for single-cell rna-seq clustering and annotation. *Genes*, 11(7):792, 2020.
- [87] Thomas M Chen, Juan Carlos Sanchez-Aarnoutse, and John Buford. Petri net modeling of cyber-physical attacks on smart grid. *IEEE Transactions on smart grid*, 2(4):741–749, 2011.

- [88] Yixin Chen, Guozhu Dong, Jiawei Han, Benjamin W Wah, and Jianyoung Wang. Multi-dimensional regression analysis of time-series data streams. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 323–334. Elsevier, 2002.
- [89] Agnese Chiatti, Mu Jung Cho, Anupriya Gagneja, Xiao Yang, Miriam Brinberg, Katie Roehrick, Sagnik Ray Choudhury, Nilam Ram, Byron Reeves, and C Lee Giles. Text extraction and retrieval from smartphone screenshots: Building a repository for life in media. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 948–955, 2018.
- [90] Giovanni Chiola, Marco Ajmone Marsan, Gianfranco Balbo, and Gianni Conte. Generalized stochastic petri nets: A definition at the net level and its implications. *IEEE Transactions on software engineering*, 19(2):89–107, 1993.
- [91] Seokwoo Choi. Api deobfuscator: Identifying runtime-obfuscated api calls via memory access analysis. *Black Hat Asia*, 2015.
- [92] Stanley Chow, Philip Eisen, Harold Johnson, and Paul C Van Oorschot. White-box cryptography and an aes implementation. In *International Workshop on Selected Areas in Cryptography*, pages 250–270. Springer, 2002.
- [93] Kevin Chung. Live lesson: Lowering the barriers to capture the flag administration and participation. In *2017 USENIX Workshop on Advances in Security Education (ASE 17)*, 2017.
- [94] Christian Collberg. The tigress c obfuscator.
- [95] Christian Collberg, Jack Davidson, Roberto Giacobazzi, Yuan Xiang Gu, Amir Herzberg, and Fei-Yue Wang. Toward digital asset protection. *IEEE Intelligent Systems*, 26(6):8–13, 2011.
- [96] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.
- [97] Christian S. Collberg and Clark Thomborson. Watermarking, tamper-proofing, and obfuscation-tools for software protection. *IEEE Transactions on software engineering*, 28(8):735–746, 2002.
- [98] Luigi Coniglio. Combining program synthesis and symbolic execution to deobfuscate binary code. Master’s thesis, University of Twente, 2019.

- [99] Gregory Conti, Erik Dean, Matthew Sinda, and Benjamin Sangster. Visual reverse engineering of binary and data files. In *International Workshop on Visualization for Computer Security*, pages 1–17. Springer, 2008.
- [100] Bart Coppens, Bjorn De Sutter, and Jonas Maebe. Feedback-driven binary code diversification. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):1–26, 2013.
- [101] George C Dalton, Robert F Mills, John M Colombi, Richard A Raines, et al. Analyzing attack trees using generalized stochastic petri nets. In *Information Assurance Workshop*, pages 116–123. IEEE, 2006.
- [102] Anusha Damodaran, Fabio Di Troia, Corrado Aaron Visaggio, Thomas H Austin, and Mark Stamp. A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques*, 13(1):1–12, 2017.
- [103] Stamatia Dasiopoulou, Eirini Giannakidou, Georgios Litos, Polyxeni Malasioti, and Yiannis Kompatsiaris. A survey of semantic image and video annotation tools. In *Knowledge-driven multimedia information extraction and ontology evolution*, pages 196–239. Springer, 2011.
- [104] Robin David. *Formal approaches for automatic deobfuscation and reverse-engineering of protected codes*. PhD thesis, Université de Lorraine, 2017.
- [105] Javier de San Pedro and Jordi Cortadella. Mining structured petri nets for the visualization of process behavior. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 839–846, 2016.
- [106] Bjorn De Sutter, Cataldo Basile, Mariano Ceccato, Paolo Falcarin, Michael Zunke, Brecht Wyseur, and Jerome d’Annoville. The aspire framework for software protection. In *Proceedings of the 2016 ACM Workshop on Software PROtection*, pages 91–92, 2016.
- [107] Bjorn De Sutter, Christian Collberg, Mila Dalla Preda, and Brecht Wyseur. Software protection decision support and evaluation methodologies (dagstuhl seminar 19331). In *Dagstuhl Reports*, volume 9. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
- [108] Bjorn De Sutter, Paolo Falcarin, Brecht Wyseur, Cataldo Basile, Mariano Cecato, Jerome d’Annoville, and Michael Zunke. A reference architecture for

- software protection. In *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 291–294. IEEE, 2016.
- [109] Cécile Delerablée, Tancrède Lepoint, Pascal Paillier, and Matthieu Rivain. White-box security notions for symmetric encryption schemes. In *International Conference on Selected Areas in Cryptography*, pages 247–264. Springer, 2013.
 - [110] Peter J Denning. The locality principle. In *Communication Networks And Computer Systems: A Tribute to Professor Erol Gelenbe*, pages 43–67. World Scientific, 2006.
 - [111] Asha Devi and Gaurav Aggarwal. Manual unpacking of upx packed executable using ollydbg and importrec. *IOSR Journal of Computer Engineering*, 16(1):71–77, 2014.
 - [112] Sabin Devkota. *Visualizing Control Flow Graphs*. PhD thesis, The University of Arizona, 2021.
 - [113] Rinku Dewri, Nayot Poolsappasit, Indrajit Ray, and Darrell Whitley. Optimal security hardening using multi-objective optimization on attack tree models of networks. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 204–213, 2007.
 - [114] Rinku Dewri, Indrajit Ray, Nayot Poolsappasit, and Darrell Whitley. Optimal security hardening on attack tree models of networks: a cost-benefit analysis. *International Journal of Information Security*, 11(3):167–188, 2012.
 - [115] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.
 - [116] Yevgeniy Dodis and Adam Smith. Correcting errors without leaking partial information. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 654–663, 2005.
 - [117] Chris Eagle. *The IDA pro book*. no starch press, 2011.
 - [118] Chris Eagle and Kara Nance. *The Ghidra Book: The Definitive Guide*. no starch press, 2020.
 - [119] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. Smart: Secure and minimal architecture for (establishing dynamic) root of trust. In *Ndss*, volume 12, pages 1–15, 2012.

- [120] Levent Ertaul and Suma Venkatesh. Jhide-a tool kit for code obfuscation. In *IASTED Conf. on Software Engineering and Applications*, pages 133–138, 2004.
- [121] Paul Fahn. Frequently asked questions about today’s cryptography. *Citeseer*, 1992.
- [122] Gregory Falco, Arun Viswanathan, and Andrew Santangelo. Cubesat security attack tree analysis. In *8th IEEE International Conference On Space Mission Challenges for Information Technology*, 2021.
- [123] Hui Fang, Yongdong Wu, Shuhong Wang, and Yin Huang. Multi-stage binary code obfuscation using improved virtual machine. In *International Conference on Information Security*, pages 168–181. Springer, 2011.
- [124] Eraldo R Fernandes and Ulf Brefeld. Learning from partially annotated sequences. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 407–422. Springer, 2011.
- [125] Sue Fitzgerald, Beth Simon, and Lynda Thomas. Strategies that students use to trace code: an analysis based in grounded theory. In *Proceedings of the first international workshop on Computing education research*, pages 69–80, 2005.
- [126] Mathieu Fourment and Michael R Gillings. A comparison of common programming languages used in bioinformatics. *BMC bioinformatics*, 9(1):1–9, 2008.
- [127] Marcel Frigault, Lingyu Wang, Anoop Singhal, and Sushil Jajodia. Measuring network security using dynamic bayesian network. In *Proceedings of the 4th ACM workshop on Quality of protection*, pages 23–30, 2008.
- [128] Thomas MJ Fruchterman and Edward M Reingold. Graph drawing by force-directed placement. *Software: Practice and experience*, 21(11):1129–1164, 1991.
- [129] Kazuhide Fukushima, Shinsaku Kiyomoto, and Toshiaki Tanaka. Obfuscation mechanism in conjunction with tamper-proof module. In *2009 International Conference on Computational Science and Engineering*, volume 2, pages 665–670. IEEE, 2009.

- [130] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. *SIAM Journal on Computing*, 45(3):882–929, 2016.
- [131] Saverio Giallorenzo, Jacopo Mauro, Martin Gyde Poulsen, and Filip Siroky. Virtualization costs: benchmarking containers and virtual machines against bare-metal. *SN Computer Science*, 2(5):1–20, 2021.
- [132] Nicolas Girard, Guillaume Charpiat, and Yuliya Tarabalka. Noisy supervision for correcting misaligned cadaster maps without perfect ground truth data. In *IGARSS 2019-2019 IEEE International Geoscience and Remote Sensing Symposium*, pages 10103–10106. IEEE, 2019.
- [133] Shafi Goldwasser and Yael Tauman Kalai. On the impossibility of obfuscation with auxiliary input. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS’05)*, pages 553–562. IEEE, 2005.
- [134] Shafi Goldwasser and Guy N Rothblum. On best-possible obfuscation. In *Theory of Cryptography Conference*, pages 194–213. Springer, 2007.
- [135] Louis Goubin, Pascal Paillier, Matthieu Rivain, and Junwei Wang. How to reveal the secrets of an obscure white-box implementation. *Journal of Cryptographic Engineering*, 10(1):49–66, 2020.
- [136] Louis Goubin, Matthieu Rivain, and Junwei Wang. Defeating state-of-the-art white-box countermeasures with advanced gray-box attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):454–482, 2020.
- [137] Christina Goulding. Grounded theory: the missing methodology on the interpretivist agenda. *Qualitative Market Research: an international journal*, 1998.
- [138] Vipul Goyal, Yuval Ishai, Amit Sahai, Ramarathnam Venkatesan, and Akshay Wadia. Founding cryptography on tamper-proof hardware tokens. In *Theory of Cryptography Conference*, pages 308–326. Springer, 2010.
- [139] Katie R Green, Tershia Pinder-Grover, and Joanna Mirecki Millunchick. Impact of screencast technology: Connecting the perception of usefulness and the reality of performance. *Journal of Engineering Education*, 2012.

- [140] Liang Gu, Xuhua Ding, Robert Huijie Deng, Bing Xie, and Hong Mei. Remote attestation on program execution. In *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pages 11–20, 2008.
- [141] Baris Guc, Michael May, Yucel Saygin, and Christine Körner. Semantic annotation of gps trajectories. In *11th AGILE international conference on geographic information science*, volume 38, pages 1–9, 2008.
- [142] Joey Hagedorn, Joshua Hailpern, and Karrie G Karahalios. Vcode and vdata: illustrating a new framework for supporting the video annotation workflow. In *Proceedings of the working conference on Advanced visual interfaces*, pages 317–321, 2008.
- [143] Norman Hänsch, Andrea Schankin, Mykolai Protsenko, Felix Freiling, and Zinaida Benenson. Programming experience might not help in comprehending obfuscated source code efficiently. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, pages 341–356, 2018.
- [144] Md Shariful Haque and Travis Atkison. An evolutionary approach of attack graph to attack tree conversion. *International Journal of Computer Network and Information Security*, 9(11):1, 2017.
- [145] Helen Heath and Sarah Cowley. Developing a grounded theory approach: a comparison of glaser and strauss. *International journal of nursing studies*, 41(2):141–150, 2004.
- [146] Guy Helmer, Johnny Wong, Mark Slagell, Vasant Honavar, Les Miller, Yanxin Wang, Xia Wang, and Natalia Stakhanova. Software fault tree and coloured petri net-based specification, design and implementation of agent-based intrusion detection systems. *International Journal of Information and Computer Security*, 1(1-2):109–142, 2007.
- [147] Wayne C Henry. Analytic provenance for software reverse engineers. *Air Force Institute of Technology*, 2020.
- [148] Cormac Herley and Paul C Van Oorschot. Sok: Science, security and the elusive goal of security as a scientific pursuit. In *2017 IEEE symposium on security and privacy (SP)*, pages 99–120. IEEE, 2017.
- [149] Jaap-Henk Hoepman and Bart Jacobs. Increased security through open source. *Communications of the ACM*, 50(1):79–83, 2007.

- [150] Dennis Hofheinz, John Malone-Lee, and Martijn Stam. Obfuscation for cryptographic purposes. In *Theory of Cryptography Conference*, pages 214–232. Springer, 2007.
- [151] William Holder, J Todd McDonald, and Todd R Andel. Evaluating optimal phase ordering in obfuscation executives. In *Proceedings of the 7th Software Security, Protection, and Reverse Engineering/Software Security and Protection Workshop*, pages 1–12, 2017.
- [152] John Homer, Ashok Varikuti, Xinming Ou, and Miles A McQueen. Improving attack graph visualization through data reduction and attack grouping. In *International Workshop on Visualization for Computer Security*, pages 68–79. Springer, 2008.
- [153] Ross Horne, Sjouke Mauw, and Alwen Tiu. Semantics for specialising attack trees based on linear logic. *Fundamenta Informaticae*, 153(1-2):57–86, 2017.
- [154] Máté Horváth. Survey on cryptographic obfuscation. *IACR Cryptol. ePrint Arch.*, 2015:412, 2015.
- [155] Shohreh Hosseinzadeh, Sampsa Rauti, Samuel Laurén, Jari-Matti Mäkelä, Johannes Holvitie, Sami Hyrynsalmi, and Ville Leppänen. Diversification and obfuscation techniques for software security: A systematic literature review. *Information and Software Technology*, 104:72–93, 2018.
- [156] Ting-Wei Hou, Hsiang-Yang Chen, and Ming-Hsiu Tsai. Three control flow obfuscation methods for java software. *IEE Proceedings-Software*, 153(2):80–86, 2006.
- [157] Yu Huang, Xinyu Liu, Ryan Krueger, Tyler Santander, Xiaosu Hu, Kevin Leach, and Westley Weimer. Distilling neural representations of data structure manipulation using fmri and fnirs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 396–407. IEEE, 2019.
- [158] Oscar H Ibarra and Shlomo Moran. Probabilistic algorithms for deciding equivalence of straight-line programs. *Journal of the ACM (JACM)*, 30(1):217–228, 1983.
- [159] Amjad Ibrahim, Stevica Bozhinoski, and Alexander Pretschner. Attack graph generation for microservice architecture. In *Proceedings of the 34th ACM/SIGAPP symposium on applied computing*, pages 1235–1242, 2019.

- [160] Terrance R Ingoldsby. Attack tree-based threat risk analysis. *Amenaza Technologies Limited*, pages 3–9, 2010.
- [161] Kyle Ingols, Richard Lippmann, and Keith Piwowarski. Practical attack graph generation for network defense. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 121–130. IEEE, 2006.
- [162] Thanapong Intharah, Daniyar Turmukhambetov, and Gabriel J Brostow. Help, it looks confusing: Gui task automation through demonstration and follow-up questions. In *Proceedings of the 22Nd International Conference on Intelligent User Interfaces*, pages 233–243, 2017.
- [163] Maki Inui and Tetsuya Izu. Current status on elliptic curve discrete logarithm problem. In *International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 537–539. Springer, 2018.
- [164] Irdeto. The company, Dec 2021.
- [165] Marieta Georgieva Ivanova, Christian W Probst, René Rydhof Hansen, and Florian Kammüller. Attack tree generation by policy invalidation. In *IFIP International Conference on Information Security Theory and Practice*, pages 249–259. Springer, 2015.
- [166] Abiodun Iwayemi and Chi Zhou. Saraa: Semi-supervised learning for automated residential appliance annotation. *IEEE Transactions on Smart Grid*, 8(2):779–786, 2015.
- [167] Alejandro Jaimes and Nicu Sebe. Multimodal human–computer interaction: A survey. *Computer vision and image understanding*, 108(1-2):116–134, 2007.
- [168] Aayush Jain, Huijia Lin, and Amit Sahai. Indistinguishability obfuscation from well-founded assumptions. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 60–73, 2021.
- [169] Sainadh Jamalpur, Yamini Sai Navya, Perla Raja, Gampala Tagore, and G Rama Koteswara Rao. Dynamic malware analysis using cuckoo sandbox. In *2018 Second international conference on inventive communication and computational technologies (ICICCT)*, pages 1056–1060. IEEE, 2018.
- [170] Ravi Jhawar, Karim Lounis, Sjouke Mauw, and Yunior Ramírez-Cruz. Semi-automatically augmenting attack trees using an annotated attack tree library.

- In *International Workshop on Security and Trust Management*, pages 85–101. Springer, 2018.
- [171] Harshvardhan P Joshi, Aravindhan Dhanasekaran, and Rudra Dutta. Trading off a vulnerability: does software obfuscation increase the risk of rop attacks. *Journal of Cyber Security and Mobility*, pages 305–324, 2015.
 - [172] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michelin. Obfuscator-LVM – Software Protection for the Masses. In Brecht Wyseur, editor, *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO’15, Firenze, Italy, May 19th, 2015*, pages 3–9, 2015.
 - [173] Anatoli Kalysch, Johannes Götzfried, and Tilo Müller. Vmattack: Deobfuscating virtualization-based packed binaries. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, page 2. ACM, 2017.
 - [174] Yuichiro Kanzaki, Akito Monden, and Christian Collberg. Code artificiality: a metric for the code stealth based on an n-gram model. In *2015 IEEE/ACM 1st International Workshop on Software Protection*, pages 31–37. IEEE, 2015.
 - [175] Fazle Karim, Somshubra Majumdar, Houshang Darabi, and Shun Chen. Lstm fully convolutional networks for time series classification. *IEEE access*, 6:1662–1669, 2017.
 - [176] Sanjeev Karmakar and Siddhartha Choubey. Pragmatic implementation of rsa-1024 cryptography. *Communications*, 3:19–27, 2015.
 - [177] Khaled Karray, Jean-Luc Danger, Sylvain Guilley, and M Abdelaziz Elaabid. Attack tree construction and its application to the connected vehicle. In *Cyber-Physical Systems Security*, pages 175–190. Springer, 2018.
 - [178] Vamsee Kasavajhala. Solid state drive vs. hard disk drive price and performance study. *Proc. Dell Tech. White Paper*, pages 8–9, 2011.
 - [179] Kerem Kaynar. A taxonomy for attack graph generation and usage in network security. *Journal of Information Security and Applications*, 29:27–56, 2016.
 - [180] Kerem Kaynar and Fikret Sivrikaya. Distributed attack graph generation. *IEEE Transactions on Dependable and Secure Computing*, 13(5):519–532, 2015.

- [181] Abdulkarim Khormi, Mohammad Alahmadi, and Sonia Haiduc. A study on the accuracy of ocr engines for source code transcription from programming screencasts. In *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020.
- [182] Byung-Ik Kim, Chae-Tae Im, and Hyun-Chul Jung. Suspicious malicious web site detection with strength analysis of a javascript obfuscation. *International Journal of Advanced Science and Technology*, 26:19–32, 2011.
- [183] Thorsten Kleinjung, Joppe W Bos, Arjen K Lenstra, Dag Arne Osvik, Kazumaro Aoki, Scott Contini, Jens Franke, Emmanuel Thomé, Pascal Jermini, Michela Thiémard, et al. A heterogeneous computing environment to solve the 768-bit rsa challenge. *Cluster Computing*, 15(1):53–68, 2012.
- [184] Neal Koblitz and Alfred Menezes. Another look at security definitions. *Cryptography ePrint Archive*, 2011.
- [185] Neal Koblitz and Alfred J Menezes. Another look at " provable security". *Journal of Cryptology*, 20(1):3–37, 2007.
- [186] Patrick Kochberger, Sebastian Schrittwieser, Stefan Schweighofer, Peter Kieseberg, and Edgar Weippl. Sok: Automatic deobfuscation of virtualization-protected applications. In *The 16th International Conference on Availability, Reliability and Security*, pages 1–15, 2021.
- [187] Igor Kotenko and Andrey Chechulin. A cyber attack modeling and impact assessment framework. In *2013 5th International Conference on Cyber Conflict (CYCON 2013)*, pages 1–24. IEEE, 2013.
- [188] I Kotuliak, P Rybár, and P Trúchly. Performance comparison of ipsec and tls based vpn technologies. In *2011 9th International Conference on Emerging eLearning Technologies and Applications (ICETA)*, pages 217–221. IEEE, 2011.
- [189] Jules Kouatchou and Alexander Medema. Basic comparison of python, julia, matlab, idl and java (2018 edition). *Modeling Guru–National Aeronautics and Space Administration, USA*, 2018.
- [190] Aleksandrina Kovacheva. Efficient code obfuscation for android. In *International Conference on Advances in Information Technology*, pages 104–119. Springer, 2013.

- [191] Kyle Kafka, Aditya Khosla, Petr Kellnhofer, Harini Kannan, Suchendra Bhandarkar, Wojciech Matusik, and Antonio Torralba. Eye tracking for everyone. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2176–2184, 2016.
- [192] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *USENIX security Symposium*, volume 13, pages 18–18, 2004.
- [193] Stela Kucek and Maria Leitner. An empirical survey of functions and configurations of open-source capture the flag (ctf) environments. *Journal of Network and Computer Applications*, 151:102470, 2020.
- [194] John Lai and Nicole O Widmar. Revisiting the digital divide in the covid-19 era. *Applied economic perspectives and policy*, 43(1):458–464, 2021.
- [195] Harjinder Singh Lallie, Kurt Debattista, and Jay Bal. An empirical evaluation of the effectiveness of attack graphs and fault trees in cyber-attack perception. *IEEE Transactions on Information Forensics and Security*, 13(5):1110–1122, 2017.
- [196] Harjinder Singh Lallie, Kurt Debattista, and Jay Bal. Evaluating practitioner cyber-security attack graph configuration preferences. *Computers & Security*, 79:117–131, 2018.
- [197] Harjinder Singh Lallie, Kurt Debattista, and Jay Bal. A review of attack graph and attack tree visual syntax in cyber security. *Computer Science Review*, 35:100219, 2020.
- [198] Andrew Lamoureux. Binary ninja debugger showcase.
- [199] Christina Latsou, Sarah Dunnett, and Lisa Jackson. A new methodology for automated petri net generation: Method application. *Loughborough University*, 2019.
- [200] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

- [201] Samuel Laurén, Petteri Mäki, Sampsaa Rauti, Shohreh Hosseinzadeh, Sami Hyrynsalmi, and Ville Leppänen. Symbol diversification of linux binaries. In *World Congress on Internet Security (WorldCIS-2014)*, pages 74–79. IEEE, 2014.
- [202] Samuel Laurén, Sampsaa Rauti, and Ville Leppänen. A survey on application sandboxing techniques. In *Proceedings of the 18th International Conference on Computer Systems and Technologies*, pages 141–148, 2017.
- [203] NY Louis Lee and PN Johnson-Laird. A theory of reverse engineering and its application to boolean systems. *Journal of Cognitive Psychology*, 25(4):365–389, 2013.
- [204] Jerzy Letkowski. Doing database design with mysql. *Journal of Technology Research*, 6:1, 2015.
- [205] Kunpeng Li, Chen Fang, Zhaowen Wang, Seokhwan Kim, Hailin Jin, and Yun Fu. Screencast tutorial video understanding. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 12526–12535, 2020.
- [206] Toby Jia-Jun Li. *A Multi-Modal Intelligent Agent that Learns from Demonstrations and Natural Language Instructions*. PhD thesis, University of California San Diego, 2021.
- [207] Xinlei Li and Di Li. A network attack model based on colored petri net. *Journal of networks*, 9(7):1883, 2014.
- [208] Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, and Ke Yi. Tree indexing on solid state drives. *Proceedings of the VLDB Endowment*, 3(1-2):1195–1206, 2010.
- [209] Clifford Liem, Yuan Xiang Gu, and Harold Johnson. A compiler-based infrastructure for software-protection. In *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, pages 33–44, 2008.
- [210] Jurgita Lieponienė. Recent trends in database technology. *Baltic Journal of Modern Computing*, 8(4):551–559, 2020.

- [211] Peter Likarish, Eunjin Jung, and Insoon Jo. Obfuscated malicious javascript detection using classification techniques. In *2009 4th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 47–54. IEEE, 2009.
- [212] Fangzhen Lin. Reducing strong equivalence of logic programs to entailment in classical propositional logic. In *International Conference on Principles of Knowledge Representation and Reasoning, Toulouse, France*, 2002.
- [213] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299, 2003.
- [214] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security, CCS '03*, pages 290–299. ACM, 2003.
- [215] Richard Paul Lippmann and Kyle William Ingols. An annotated review of past papers on attack graphs. *Massachusetts Inst of Tech Lexington Lincoln Lab*, 2005.
- [216] Steven A Lloyd and Chuck L Robertson. Screencast tutorials enhance student learning of statistics. *Teaching of Psychology*, 39(1):67–71, 2012.
- [217] Weiyun Lu, Bahman Sistany, Amy Felty, and Philip Scott. Towards formal verification of program obfuscation. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 635–644. IEEE, 2020.
- [218] Patrick Luckett, J Todd McDonald, and William Bradley Glisson. Attack-graph threat modeling assessment of ambulatory medical devices. *arXiv preprint arXiv:1709.05026*, 2017.
- [219] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In Vivek Sarkar and Mary W. Hall, editors, *PLDI*, pages 190–200. ACM, 2005.
- [220] Saturnino Luz and Masood Masoodian. Comparing static gantt and mosaic charts for visualization of task schedules. In *2011 15th International Conference on Information Visualisation*, pages 182–187. IEEE, 2011.

- [221] Benjamin Lynn, Manoj Prabhakaran, and Amit Sahai. Positive results and techniques for obfuscation. In *International conference on the theory and applications of cryptographic techniques*, pages 20–39. Springer, 2004.
- [222] Laura MacLeod, Andreas Bergen, and Margaret-Anne Storey. Documenting and sharing software knowledge using screencasts. *Empirical Software Engineering*, 22(3):1478–1507, 2017.
- [223] Pieter Maene, Johannes Götzfried, Ruan De Clercq, Tilo Müller, Felix Freiling, and Ingrid Verbauwhede. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Transactions on Computers*, 67(3):361–374, 2017.
- [224] Dominik Maier and Lukas Seidel. Jmpscare: Introspection for binary-only fuzzing. In *Workshop on Binary Analysis Research (BAR)*, volume 2021, page 21, 2021.
- [225] Ritu Malik and Rupali Syal. Performance analysis of ip security vpn. *International Journal of Computer Applications*, 8(4):0975, 2010.
- [226] Jean-Yves Marion and Daniel Reynaud. Dynamic binary instrumentation for deobfuscation and unpacking. In *Depth Security Conference*, 2009.
- [227] M Ajmone Marsan, Gianfranco Balbo, Gianni Conte, Susanna Donatelli, and Giuliana Franceschinis. Modelling with generalized stochastic petri nets. *ACM SIGMETRICS performance evaluation review*, 26(2):2, 1998.
- [228] Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Conqueror: tamper-proof code execution on legacy systems. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 21–40. Springer, 2010.
- [229] Vasileios Mavroeidis and Siri Bromander. Cyber threat intelligence model: an evaluation of taxonomies, sharing standards, and ontologies within cyber threat intelligence. In *2017 European Intelligence and Security Informatics Conference (EISIC)*, pages 91–98. IEEE, 2017.
- [230] James P McDermott. Attack net penetration testing. In *Proceedings of the 2000 workshop on New security paradigms*, pages 15–21, 2001.

- [231] Stephen McLaughlin, Dmitry Podkuiko, and Patrick McDaniel. Energy theft in the advanced metering infrastructure. In *International Workshop on Critical Information Infrastructures Security*, pages 176–187. Springer, 2009.
- [232] Sean McManus and Mike Cook. *Raspberry Pi for dummies*. John Wiley & Sons, 2021.
- [233] Daniel Mercier, Aziem Chawdhary, and Richard Jones. dynstruct: An automatic reverse engineering tool for structure recovery and memory use analysis. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 497–501. IEEE, 2017.
- [234] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. Probabilistic disassembly. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1187–1198. IEEE, 2019.
- [235] Jimmy Moore, Pascal Goffin, Miriah Meyer, Philip Lundrigan, Neal Patwari, Katherine Sward, and Jason Wiese. Managing in-home environments through sensing, annotating, and visualizing air quality data. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2(3):1–28, 2018.
- [236] Azqa Nadeem, Sicco Verwer, Stephen Moskal, and Shanchieh Jay Yang. Alert-driven attack graph generation using s-pdfa. *IEEE Transactions on Dependable and Secure Computing*, 2021.
- [237] Jasvir Nagra and Christian Collberg. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Pearson Education, 2009.
- [238] Susanta Nanda, Wei Li, Lap-Chung Lam, and Tzi-cker Chiueh. Bird: Binary interpretation using runtime disassembly. In *International Symposium on Code Generation and Optimization (CGO’06)*, pages 12–pp. IEEE, 2006.
- [239] Payam Mahmoudi Nasr and Ali Yazdian Varjani. Petri net model of insider attacks in scada system. In *2014 11th International ISC Conference on Information Security and Cryptology*, pages 55–60. IEEE, 2014.
- [240] Minh Hai Nguyen, Thien Binh Nguyen, Thanh Tho Quan, and Mizuhito Ogawa. A hybrid approach for control flow graph construction from binary code. In *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, volume 2, pages 159–164. IEEE, 2013.

- [241] Sam Nguyen. *Automated attack tree generation and evaluation: systemization of knowledge*. PhD thesis, Massachusetts Institute of Technology, 2020.
- [242] Tuan Anh Nguyen and Christoph Csallner. Reverse engineering mobile application user interfaces with remaui (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 248–259. IEEE, 2015.
- [243] Steven Noel and Sushil Jajodia. Managing attack graph complexity through visual hierarchical aggregation. In *Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*, pages 109–118, 2004.
- [244] Leo Obrst, Penny Chase, and Richard Markeloff. Developing an ontology of the cyber security domain. In *STIDS*, pages 49–56. Citeseer, 2012.
- [245] Michael Opdenacker. Embedded linux size reduction techniques. In *Embedded Linux Conference*, 2017.
- [246] Xinning Ou, Wayne F Boyer, and Miles A McQueen. A scalable approach to attack graph generation. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 336–345, 2006.
- [247] Anthony Overmars. Survey of rsa vulnerabilities. In *Modern Cryptography- Current Challenges and Solutions*. IntechOpen, 2019.
- [248] Anthony Overmars and Sitalakshmi Venkatraman. New semi-prime factorization and application in large rsa key attacks. *Journal of Cybersecurity and Privacy*, 1(4):660–674, 2021.
- [249] Philippe Palanque, Rémi Bastide, and Valérie Sengès. Validating interactive system design through the verification of formal task and system models. In *IFIP International Conference on Engineering for Human-Computer Interaction*, pages 189–212. Springer, 1995.
- [250] Andriy Panchenko, Fabian Lanze, and Thomas Engel. Improving performance and anonymity in the tor network. In *2012 IEEE 31st International Performance Computing and Communications Conference (IPCCC)*, pages 1–10. IEEE, 2012.
- [251] Gabriele Paolacci, Jesse Chandler, and Panagiotis G Ipeirotis. Running experiments on amazon mechanical turk. *Judgment and Decision making*, 5(5):411–419, 2010.

- [252] Sung Hyun Park and Bong Nam Noh. Guided symbolic execution in real-world binary program. In *Information Science and Applications*, pages 387–396. Springer, 2020.
- [253] Roy D Pea and D Midian Kurland. On the cognitive effects of learning computer programming. *New ideas in psychology*, 2(2):137–168, 1984.
- [254] Ugo Piazzalunga, Paolo Salvaneschi, Francesco Balducci, Pablo Jacomuzzi, and Cristiano Moroncelli. Security strength measurement for dongle-protected software. *IEEE Security & Privacy*, 5(6):32–40, 2007.
- [255] Ludovic Piètre-Cambacédès and Marc Bouissou. Beyond attack trees: dynamic security modeling with boolean logic driven markov processes (bdmp). In *2010 European Dependable Computing Conference*, pages 199–208. IEEE, 2010.
- [256] Suporn Pongnumkul, Mira Dontcheva, Wilmot Li, Jue Wang, Lubomir Bourdev, Shai Avidan, and Michael F Cohen. Pause-and-play: automatically linking screencast video tutorials with applications. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 135–144, 2011.
- [257] Jon Postel and Joyce Reynolds. File transfer protocol. *STD 9, RFC 959, October*, 1985.
- [258] R Prakash, PP Amritha, and M Sethumadhavan. Opaque predicate detection by static analysis of binary executables. In *International Symposium on Security in Computing and Communication*, pages 250–258. Springer, 2017.
- [259] Lutz Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, 2000.
- [260] Srdjan Pudar, Govindarasu Manimaran, and Chen-Ching Liu. Penet: A practical method and tool for integrated modeling of security attacks and countermeasures. *Computers & Security*, 28(8):754–771, 2009.
- [261] Jing Qiu, Babak Yadegari, Brian Johannesmeyer, Saumya Debray, and Xiaohong Su. Identifying and understanding self-checksumming defenses in software. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 207–218, 2015.

- [262] Jason Raber and Eric Laspe. Deobfuscator: An automated approach to the identification and removal of code obfuscation. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pages 275–276. IEEE, 2007.
- [263] Inge Alexander Raknes, Bjørn Fjukstad, and Lars Ailo Bongo. nsroot: Minimalist process isolation tool implemented with linux namespaces. *arXiv preprint arXiv:1609.03750*, 2016.
- [264] Daniel Ramsbrock, Robin Berthier, and Michel Cukier. Profiling attacker behavior following ssh compromises. In *37th Annual IEEE/IFIP international conference on dependable systems and networks (DSN'07)*, pages 119–124. IEEE, 2007.
- [265] David Rasch and Randal C Burns. In-place rsync: File synchronization for mobile and wireless devices. In *USENIX Annual Technical Conference, FREENIX Track*, volume 100, 2003.
- [266] Sampsa Rauti, Samuel Laurén, Petteri Mäki, Joni Uitto, Samuli Laato, and Ville Leppänen. Internal interface diversification as a method against malware. *Journal of Cyber Security Technology*, 5(1):15–40, 2021.
- [267] Sylvestre-Alvise Rebuffi, Sébastien Ehrhardt, Kai Han, Andrea Vedaldi, and Andrew Zisserman. Semi-supervised learning with scarce annotations. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 762–763, 2020.
- [268] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271, 2017.
- [269] Amjad Rehman, Sultan Alqahtani, Ayman Altameem, and Tanzila Saba. Virtual machine security challenges: case studies. *International Journal of Machine Learning and Cybernetics*, 5(5):729–742, 2014.
- [270] Tobias Roehm, Nigar Gurbanova, Bernd Bruegge, Christophe Joubert, and Walid Maalej. Monitoring user interactions for supporting failure reproduction. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 73–82. IEEE, 2013.
- [271] Roman Rohleder. Hands-on ghidra-a tutorial about the software reverse engineering framework. In *Proceedings of the 3rd ACM Workshop on Software Protection*, pages 77–78, 2019.

- [272] Rolf Rolles. Unpacking virtualization obfuscators. In *3rd USENIX Workshop on Offensive Technologies.(WOOT)*, 2009.
- [273] Sharhida Zawani Saad and Udo Kruschwitz. Exploiting click logs for adaptive intranet navigation. In *European Conference on Information Retrieval*, pages 792–795. Springer, 2013.
- [274] Cagri Sahin, Mian Wan, Philip Tornquist, Ryan McKenna, Zachary Pearson, William GJ Halfond, and James Clause. How does code obfuscation impact energy usage? *Journal of Software: Evolution and Process*, 28(7):565–588, 2016.
- [275] Martin Salfer and Claudia Eckert. Attack surface and vulnerability assessment of automotive electronic control units. In *2015 12th International Joint Conference on e-Business and Telecommunications (ICETE)*, volume 4, pages 317–326. IEEE, 2015.
- [276] Stephan Salinger, Laura Plonka, and Lutz Prechelt. A coding scheme development methodology using grounded theory for qualitative analysis of pair programming. *Human Technology: An Interdisciplinary Journal on Humans in ICT Environments*, 2008.
- [277] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. Deobfuscation of vm based software protection. In *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC'17)*, 2017.
- [278] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. Symbolic deobfuscation: From virtualized code back to the original. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 372–392. Springer, 2018.
- [279] Spyridon Samonas and David Coss. The cia strikes back: Redefining confidentiality, integrity and availability in security. *Journal of Information System Security*, 10(3), 2014.
- [280] Ramadass Sathya, Annamma Abraham, et al. Comparison of supervised and unsupervised learning algorithms for pattern classification. *International Journal of Advanced Research in Artificial Intelligence*, 2(2):34–38, 2013.
- [281] Florent Saudel and Jonathan Salwan. Triton: A dynamic symbolic execution framework. In *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes*, pages 31–54, 2015.

- [282] Robert W Scheifler and Jim Gettys. The x window system. *ACM Transactions on Graphics (TOG)*, 5(2):79–109, 1986.
- [283] Nathan Daniel Schiele and Olga Gadyatskaya. A novel approach for attack tree to attack graph transformation: Extended version. *arXiv preprint arXiv:2110.02553*, 2021.
- [284] Sebastian Schrittweiser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Computing Surveys (CSUR)*, 49(1):1–37, 2016.
- [285] Joan Serra, Meinard Müller, Peter Grosche, and Josep Ll Arcos. Unsupervised music structure annotation by time series structure features and segment similarity. *IEEE Transactions on Multimedia*, 16(5):1229–1240, 2014.
- [286] Burr Settles. Closing the loop: Fast, interactive semi-supervised annotation with queries on features and instances. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 1467–1478, 2011.
- [287] Zohreh Sharafi, Yu Huang, Kevin Leach, and Westley Weimer. Toward an objective measure of developers’ cognitive activities. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(3):1–40, 2021.
- [288] KC Shashidhar. *Efficient automatic verification of loop and data-flow transformations by functional equivalence checking*. PhD thesis, PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven . . . , 2008.
- [289] Fei Shi, Jiajun Wang, and Zhiyong Wang. Region-based supervised annotation for semantic image retrieval. *AEU-International Journal of Electronics and Communications*, 65(11):929–936, 2011.
- [290] Gyanaranjan Shial and S Majhi. Techniques for file synchronization: a survey. *Journal of Global Research in Computer Science*, 5(11):1–4, 2014.
- [291] Min B Shrestha and Guna R Bhatta. Selecting appropriate methodological framework for time series data analysis. *The Journal of Finance and Data Science*, 4(2):71–89, 2018.
- [292] Muhammad Ali Siddiqi, Robert M Seepers, Mohammad Hamad, Vassilis Prevelakis, and Christos Strydis. Attack-tree-based threat modeling of medical implants. In *PROOFS@ CHES*, pages 32–49, 2018.

- [293] Rami Sihwail, Khairuddin Omar, and Khairul Akram Zainol Ariffin. A survey on malware analysis techniques: Static, dynamic, hybrid and memory analysis. *International Journal on Advanced Science, Engineering and Information Technology*, 8(4-2):1662, 2018.
- [294] Azzam Sleit et al. Evaluating indexeddb performance on web browsers. In *2017 8th International Conference on Information Technology (ICIT)*, pages 488–494. IEEE, 2017.
- [295] Anders Søgaard, Barbara Plank, and Dirk Hovy. Selection bias, label bias, and bias in ground truth. In *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Tutorial Abstracts*, pages 11–13, 2014.
- [296] Jacob Springer and Wu-chang Feng. Teaching with angr: A symbolic execution curriculum and {CTF}. In *2018 { USENIX } Workshop on Advances in Security Education ({ASE} 18)*, 2018.
- [297] Richard Stallman, Roland Pesch, Stan Shebs, et al. Debugging with gdb. *Free Software Foundation*, 675, 1988.
- [298] J. Stephens, B. Yadegari, C. Collberg, S. Debray, and C. Scheidegger. Probabilistic obfuscation through covert channels. In *2018 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 243–257, April 2018.
- [299] Panos Stinis. Enforcing constraints for time series prediction in supervised, unsupervised and reinforcement learning. *arXiv preprint arXiv:1905.07501*, 2019.
- [300] Qing Su, Fan He, Naiqi Wu, and Zhiyi Lin. A method for construction of software protection technology application sequence based on petri net with inhibitor arcs. *IEEE Access*, 6:11988–12000, 2018.
- [301] Kyung Sung. Trend analysis of x window used in linux. *Journal of Digital Contents Society*, 18(7):1393–1401, 2017.
- [302] Anjali J Suresh and Sriram Sankaran. Power profiling and analysis of code obfuscation for embedded devices. In *2020 IEEE 17th India Council International Conference (INDICON)*, pages 1–6. IEEE, 2020.

- [303] Iain Sutherland, George E Kalb, Andrew Blyth, and Gaius Mulley. An empirical examination of the reverse engineering process for binary files. *Computers & Security*, 25(3):221–228, 2006.
- [304] Tarja Systa. On the relationships between static and dynamic models in reverse engineering java software. In *Sixth Working Conference on Reverse Engineering (Cat. No. PR00303)*, pages 304–313. IEEE, 1999.
- [305] Mahin Talukder, Syed Islam, and Paolo Falcarin. Analysis of obfuscated code with program slicing. In *2019 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*, pages 1–7. IEEE, 2019.
- [306] Romain Tavenard, Johann Faouzi, Gilles Vandewiele, Felix Divo, Guillaume Androz, Chester Holtz, Marie Payne, Roman Yurchak, Marc Rußwurm, Kushal Kolar, et al. Tslearn, a machine learning toolkit for time series data. *J. Mach. Learn. Res.*, 21(118):1–6, 2020.
- [307] Claire Taylor and Christian Collberg. Getting RevEngE: a system for analyzing reverse engineering behavior. In *14th International Conference on Malicious and Unwanted Software "MALCON 2019" (MALCON 2019)*, Nantucket, USA, October 2019.
- [308] Clark Taylor, Pablo Arias, Jim Klopchic, Celeste Matarazzo, and Evi Dube. Ctf: State-of-the-art and building the next generation. In *2017 {USENIX} Workshop on Advances in Security Education ({ASE} 17)*. USENIX Association, 2017.
- [309] Clark Taylor and Christian Collberg. A tool for teaching reverse engineering. In *2016 USENIX Workshop on Advances in Security Education (ASE 16)*. USENIX Association, 2016.
- [310] Swizec Teller and Andrew H Hug. *Data Visualization with d3. js*, volume 78. Packt Publishing Birmingham, UK, 2013.
- [311] Zheng Tian, Weidong Wu, Shu Li, Xi Li, Yizhen Sun, and Zhongwei Chen. A security model of scada system based on attack tree. In *2019 IEEE 3rd Conference on Energy Internet and Energy System Integration (EI2)*, pages 2653–2658. IEEE, 2019.

- [312] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in {GCC} & {LLVM}. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 941–955, 2014.
- [313] Brian Tierney, Ezra Kissel, Martin Swany, and Eric Pouyoul. Efficient data transfer protocols for big data. In *2012 IEEE 8th International Conference on E-Science*, pages 1–9. IEEE, 2012.
- [314] Kris Tiri. Side-channel attack pitfalls. In *2007 44th ACM/IEEE Design Automation Conference*, pages 15–20. IEEE, 2007.
- [315] Ramtine Tofighi-Shirazi, Irina Măriuca Asăvoae, and Philippe Elbaz-Vincent. Fine-grained static detection of obfuscation transforms using ensemble-learning and semantic reasoning. In *Proceedings of the 9th Workshop on Software Security, Protection, and Reverse Engineering*, pages 1–12, 2019.
- [316] Ramtine Tofighi-Shirazi, Irina-Mariuca Asăvoae, Philippe Elbaz-Vincent, and Thanh-Ha Le. Defeating opaque predicates statically through machine learning and binary analysis. In *Proceedings of the 3rd ACM Workshop on Software Protection*, pages 3–14, 2019.
- [317] Dipty Tripathi, Ashish Kumar Maurya, Amrita Chaturvedi, and Anil Kumar Tripathi. A study of security modeling techniques for smart systems. In *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*, pages 87–92. IEEE, 2019.
- [318] Hsiao-Yu Fish Tung, Adam W Harley, William Seto, and Katerina Fragkiadaki. Adversarial inverse graphics networks: Learning 2d-to-3d lifting and image-to-image translation from unpaired supervision. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 4364–4372. IEEE, 2017.
- [319] Paul Van De Zande. The day des died. *SANS Institute*, 2001.
- [320] Ludo Van Put, Dominique Chanet, Bruno De Bus, Bjorn De Sutter, and Koen De Bosschere. Diablo: a reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology, 2005.*, pages 7–12. IEEE, 2005.
- [321] Ramanni J Veeraraghava. Security analysis of vehicle to vehicle arada locomate on board unit. *Iowa State University*, 2019.

- [322] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. Sok: Fully homomorphic encryption compilers. *arXiv preprint arXiv:2101.07078*, 2021.
- [323] Roberto Vigo, Flemming Nielson, and Hanne Riis Nielson. Automated generation of attack trees. In *2014 IEEE 27th computer security foundations symposium*, pages 337–350. IEEE, 2014.
- [324] Alessio Viticchié, Cataldo Basile, Andrea Avancini, Mariano Ceccato, Bert Abrath, and Bart Coppens. Reactive attestation: Automatic detection and reaction to software tampering attacks. In *Proceedings of the 2016 ACM workshop on software PROtection*, pages 73–84, 2016.
- [325] Alessio Viticchié, Leonardo Regano, Marco Torchiano, Cataldo Basile, Mariano Ceccato, Paolo Tonella, and Roberto Tiella. Assessment of source code obfuscation techniques. In *Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on*, pages 11–20. IEEE, 2016.
- [326] Daniel Votipka, Mary Nicole Punzalan, Seth M Rabin, Yla Tausczik, and Michelle L Mazurek. An investigation of online reverse engineering community discussions in the context of ghidra. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 1–20. IEEE, 2021.
- [327] 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 ({USENIX} Security 20)*, pages 1875–1892, 2020.
- [328] Cynthia Wagner, Alexandre Dulaunoy, Gérard Wagener, and Andras Iklody. Misp: The design and implementation of a collaborative threat intelligence sharing platform. In *Proceedings of the 2016 ACM on Workshop on Information Sharing and Collaborative Security*, pages 49–56, 2016.
- [329] Fish Wang and Yan Shoshtaishvili. Angr—the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 8–9. IEEE, 2017.
- [330] Huaijun Wang, Dingyi Fang, Ni Wang, Zhanyong Tang, Feng Chen, and Yuanxiang Gu. Method to evaluate software protection based on attack modeling. In *IEEE 10th Int'l Conf. on High Performance Computing and Communications*, pages 837–844, November 2013.

- [331] Jie Wang, Raphael C-W Phan, John N Whitley, and David J Parish. Augmented attack tree modeling of sql injection attacks. In *2010 2nd IEEE International Conference on Information Management and Engineering*, pages 182–186. IEEE, 2010.
- [332] Junwei Wang, Stefan Kölbl, Louis Goubin, Pascal Paillier, Matthieu Rivain, and Aleksei Udovenko. Ches 2021 challenge - whibox contest.
- [333] Lele Wang, Binqiang Wang, Jiangang Liu, Qiguang Miao, and Jianhui Zhang. Cuckoo-based malware dynamic analysis. *International Journal of Performance Engineering*, 15(3):772, 2019.
- [334] Shuo Wang, Jianhua Wang, Guangming Tang, and Guang Kou. A network vulnerability assessment method based on attack graph. In *2018 IEEE 4th International Conference on Computer and Communications (ICCC)*, pages 1149–1154. IEEE, 2018.
- [335] Xiaozhen Wang and Baoxu Liu. Risk assessment model building and malicious behavior detection on computer firmware. In *International Conference on High Performance Networking, Computing and Communication Systems*, pages 548–555. Springer, 2011.
- [336] Hoeteck Wee. On obfuscating point functions. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 523–532, 2005.
- [337] Eric W Weisstein. Rsa number. *Wolfram Research, Inc.*, 2003.
- [338] Steven C White and Sahra Sedigh Sarvestani. Comparison of security models: Attack graphs versus petri nets. In *Advances in Computers*, volume 94, pages 1–24. Elsevier, 2014.
- [339] Jacob Whitehill and Margo Seltzer. A crowdsourcing approach to collecting 399 tutorial videos on logarithms. *CoRR*, 2016.
- [340] Michael J Wiener. Applying software protection to white-box cryptography. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, pages 1–2, 2015.
- [341] Carina Wiesen, Nils Albartus, Max Hoffmann, Steffen Becker, Sebastian Walldat, Marc Fyrbiak, Nikol Rummel, and Christof Paar. Towards cognitive obfuscation: impeding hardware reverse engineering based on psychological insights.

- In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, pages 104–111, 2019.
- [342] Ruoyu Wu, Weiguo Li, and He Huang. An attack modeling based on hierarchical colored petri nets. In *2008 International Conference on Computer and Electrical Engineering*, pages 918–921. IEEE, 2008.
 - [343] Brecht Wyseur. White-box cryptography., 2011.
 - [344] Miguel G Xavier, Marcelo V Neves, Fabio D Rossi, Tiago C Ferreto, Timoteo Lange, and Cesar AF De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 233–240. IEEE, 2013.
 - [345] Xusheng Xiao, Xiaoyin Wang, Zhihao Cao, Hanlin Wang, and Peng Gao. Iconintent: automatic identification of sensitive ui widgets based on icon classification for android apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 257–268. IEEE, 2019.
 - [346] Peng Xie, Jason H Li, Ximming Ou, Peng Liu, and Renato Levy. Using bayesian networks for cyber security analysis. In *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pages 211–220. IEEE, 2010.
 - [347] Tao Xie, Nikolai Tillmann, Jonathan De Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 359–368. IEEE, 2009.
 - [348] Zhi Xin, Huiyu Chen, Hao Han, Bing Mao, and Li Xie. Misleading malware similarities analysis by automatic data structure obfuscation. In *International Conference on Information Security*, pages 181–195. Springer, 2010.
 - [349] Hui Xu, Yangfan Zhou, Yu Kang, and Michael R Lyu. Concolic execution on small-size binaries: Challenges and empirical study. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 181–188. IEEE, 2017.
 - [350] Tarun Yadav and Arvind Mallari Rao. Technical aspects of cyber kill chain. In *International Symposium on Security in Computing and Communication*, pages 438–452. Springer, 2015.

- [351] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A generic approach to automatic deobfuscation of executable code. In *2015 IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 674–691, May 2015.
- [352] Babak Yadegari and Saumya Debray. Symbolic execution of obfuscated code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 732–744. ACM, 2015.
- [353] Tom Yeh, Tsung-Hsiang Chang, and Robert C Miller. Sikuli: using gui screenshots for search and automation. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*, pages 183–192, 2009.
- [354] Hsiang-Fu Yu, Nikhil Rao, and Inderjit S Dhillon. Temporal regularized matrix factorization for high-dimensional time series prediction. *Advances in neural information processing systems*, 29:847–855, 2016.
- [355] Fangfang Yuan, Yanan Cao, Yanmin Shang, Yanbing Liu, Jianlong Tan, and Binxing Fang. Insider threat detection with deep neural network. In *International Conference on Computational Science*, pages 43–54. Springer, 2018.
- [356] Nezer Jacob Zaidenberg. Hardware rooted security in industry 4.0 systems. *Cyber Defence in Industry 4.0 Systems and Related Logistics and IT Infrastructures*, 51(135), 2018.
- [357] Daniel Zeman, Martin Popel, Milan Straka, Jan Hajic, Joakim Nivre, Filip Ginter, Juhani Luotolahti, Sampo Pyysalo, Slav Petrov, Martin Potthast, et al. Conll 2017 shared task: Multilingual parsing from raw text to universal dependencies. In *CoNLL 2017 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*, pages 1–19. Association for Computational Linguistics, 2017.
- [358] Rongfei Zeng, Yixin Jiang, Chuang Lin, and Xuemin Shen. Dependability analysis of control center networks in smart grid using stochastic petri nets. *IEEE Transactions on Parallel and Distributed Systems*, 23(9):1721–1730, 2012.
- [359] Gaofeng Zhang, Paolo Falcarin, Elena Gómez-Martínez, Shareeful Islam, Christophe Tartary, Bjorn De Sutter, and Jerome d’Annoville. Attack simulation based software protection assessment method. In *Cyber Security And Protection Of Digital Services (Cyber Security), 2016 International Conference On*, pages 1–8. Ieee, 2016.

- [360] Jie Zhang, David Donofrio, John Shalf, Mahmut T Kandemir, and Myoungsoo Jung. Nvmmu: A non-volatile memory management unit for heterogeneous gpu-ssd architectures. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 13–24. IEEE, 2015.
- [361] Xiaolu Zhang, Frank Breitinger, Engelbert Luechinger, and Stephen O’Shaughnessy. Android application forensics: A survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations. *Forensic Science International: Digital Investigation*, 39:301285, 2021.
- [362] Bendong Zhao, Huanzhang Lu, Shangfeng Chen, Junliang Liu, and Dongya Wu. Convolutional neural networks for time series classification. *Journal of Systems Engineering and Electronics*, 28(1):162–169, 2017.
- [363] Shuai Zhao, Xiaohong Li, Guangquan Xu, Lei Zhang, and Zhiyong Feng. Attack tree based android malware detection with hybrid analysis. In *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 380–387. IEEE, 2014.
- [364] Wentao Zhao, Pengfei Wang, and Fan Zhang. Extended petri net-based advanced persistent threat analysis model. In *Computer Engineering and Networking*, pages 1297–1305. Springer, 2014.
- [365] Wu Zhou, Xinwen Zhang, and Xuxian Jiang. Appink: watermarking android apps for repackaging deterrence. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 1–12, 2013.
- [366] Penghua Zhu, Ying Li, Tongyu Li, Wei Yang, and Yihan Xu. Gui widget detection and intent generation via image understanding. *IEEE Access*, 9:160697–160707, 2021.
- [367] Jared Ziegler. Edge of the art in vulnerability research version 4 of 4. Technical report, Two Six Labs, 2021.
- [368] Aaron Zimba, Luckson Simukonda, and Mumbi Chishimba. Demystifying ransomware attacks: reverse engineering and dynamic malware analysis of wannacry for network and information security. *Zambia ICT Journal*, 1(1):35–40, 2017.