A Qualitative Evaluation of Reverse Engineering Tool Usability

James Mattei
Tufts University
Medford, MA, USA
james.mattei@tufts.edu

Madeline McLaughlin
Tufts University
Medford, MA, USA
madeline.mclaughin@tufts.edu

Samantha Katcher
Tufts University, MITRE Corp.
Medford, MA, USA
samantha.katcher@tufts.edu

Daniel Votipka
Tufts University
Medford, MA, USA
daniel.votipka@tufts.edu

ABSTRACT

Software reverse engineering is a challenging and time consuming task. With the growing demand for reverse engineering in vulnerability discovery and malware analysis, manual reverse engineering cannot scale to meet the demand. There has been significant effort to develop automated tooling to support reverse engineers, but many reverse engineers report not using these tools. In this paper, we seek to understand whether this lack of use is an issue of usability. We performed a iterative open coding of 288 reverse engineering tools to identify common input and output methods, as well as whether the tools adhered to usability guidelines established in prior work. We found that most reverse engineering tools have limited interaction and usability support. However, usability issues vary between dynamic and static tools. Dynamic tools were less likely to provide easy-to-use interfaces, while static tools often did not allow reverse engineers to adjust the analysis. Based on our findings, we give recommendations for reverse engineering framework developers and suggest directions for future HCI research in reverse engineering.

ACM Reference Format:
James Mattei, Madeline McLaughlin, Samantha Katcher, and Daniel Votipka. 2022. A Qualitative Evaluation of Reverse Engineering Tool Usability. In *Annual Computer Security Applications Conference (ACSAC '22), December 5–9, 2022, Austin, TX, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3564625.3567993

1 INTRODUCTION

Software reverse engineering is a key component of common tasks performed by security professionals, such as vulnerability discovery and malware analysis [13, 63], [18, pp. 5-7]. With yearly increases in cybercrime, such as ransomeware [44], and the use of off-the-shelf software [29], the demand for effective reverse engineering continues to grow to analyze and defend against malware and perform necessary vulnerability review, respectively.

Reverse engineering is often a manual and artisanal craft, requiring considerable expertise and potentially heroic effort [62]. For example, Yakdan et al. observed reverse engineers (REs) required about 39 minutes on average to analyze even small decompiled programs (i.e., 150 lines of source code) [64]. Due to the complexity of reverse engineering, there is a limited population of qualified REs to perform this type of analysis [38], which cannot practically scale to meet this growing need.

To close this gap, there has been considerable effort to develop reverse engineering automation. This includes tools to simplify the reverse engineering process, e.g., by automatically renaming variable or inferring data structures [3, 4, 11, 35, 37, 64], or automating the entire process, e.g., by automatically finding exploitable vulnerabilities [6, 15, 26, 34, 56, 57]. There have also been multiple reverse engineering frameworks built that provide basic functionalities (e.g., disassembler, decompiler, debugger) and extensible plugin languages [1, 32, 50, 60]. These frameworks allow REs to develop scripts for specific functionality needs and share plugins with others [61].

While there is a growing supply of program analysis tooling, prior work showed these tools often are not used [27, 62]. Even Hex-Rays—the developers of the most popular reverse engineering framework IDA Pro, describes some of their best user-created plugins as "not exactly a walk in the park" to build and use.[1] We expect this gap occurs for two reasons: reverse engineering tools' designs do not always mesh with REs' needs and mental models and many tools are not designed for usability.

In this paper, we perform, to our knowledge, the first usability survey of reverse engineering tools. Specifically, we seek to answer the following questions:

- **RQ1**: What are the current interaction modalities for reverse engineering tools?
- **RQ2**: Do current tools fit into REs' mental models and are designed for RE usability?

RQ1 seeks to enumerate the current common modes of human-computer interaction (HCI) in reverse engineering. Because reverse engineering tools are often developed in an ad-hoc manner, this review can identify the common types of information for input and output and provide insights for interactions that could be adopted to improve user experience design. To answer this question, we

[1] https://hex-rays.com/contests_details/contest2016/#ponce

This work is licensed under a Creative Commons Attribution International 4.0 License.

ACSAC '22, December 5–9, 2022, Austin, TX, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9759-9/22/12.
https://doi.org/10.1145/3564625.3567993

performed an iterative open coding analysis [16], identifying input and output modalities of reverse engineering tools. RQ2 considers our primary question of usability. Because it would not be feasible to measure actual usability of each tool, i.e., have REs use each tool and measure performance and frustrations, we estimate usability through a heuristic evaluation. Specifically, we rely on the reverse engineering process model and usability guidelines proposed in our previous work [62]. By evaluating how a tool fits into this process model, we determine how it would fit with REs' mental models and the usability guidelines highlight the needs reported by REs. We performed a broad search for available reverse engineering tools, using existing lists [33, 46, 51, 59, 60], prior literature [5, 53, 55, 59], and reverse engineering keyword searches. We performed our open coding and heuristic evaluation on a sample of 288 reverse engineering tools.

We found that most reverse engineering tools have limited interaction and usability support. Most tools (56% 164/288) do not allow input beyond the analyzed program and many (48% 138/288) do not report their results in the context of code. The vast majority (88% 253/288) of tools have not implemented more than half of our proposed usability guidelines [62]. However, the issues with usability are not uniform across tools of differing functionality types. While dynamic tools often do not support readability improvements or interactions in the context of the code, static tools generally integrate with reverse engineering frameworks, using available APIs to produce more intuitive interactions. Conversely, static tools provide little ability for REs to tune analysis, while the use of configuration files and command-line options to adjust analysis parameters is common among dynamic tools.

We provide recommendations for reverse engineering framework developers to support improved reverse engineering usability broadly, as well as directions for future work in HCI research for reverse engineering tools.

2 BACKGROUND

Recent work in reverse engineering is beginning to consider human factors. There have been investigations into the way REs work together [45], how reverse engineering experience impacts their actions and strategies [9, 14, 20, 40, 66], how REs choose which software to investigate [2], and differences between demographic groups [14, 23, 63].

Most relevant to our work, has been the study of REs' processes when investigating a program. Through interviews with four REs, Bryant presented the first reverse engineering process model, describing reverse engineering as a sensemaking process where REs generate and test hypotheses iteratively [12]. Ceccato et al. performed an in-depth review of professional penetration testing efforts on three case study programs, along with the result of an open public challenge producing similar findings in the context of obfuscated code reverse engineering [14].

Similarly, in our prior research, we conducted 16 retrospective observational interviews with expert REs to understand their process when investigating and unfamiliar program [62]. This approach yielded a number of key insights, most notable is a three-phase model of RE analysis. In this model, REs begin with a high-level program *overview* to identify components of interest and establish initial hypotheses about the functionality. Then, reviewing the functions or blocks of interest, REs perform *subcomponent scanning*, quickly reviewing pieces of the program to refine their initial suspicious about the software. Then REs shift into *focused experimentation* when they needed to answer specific questions. REs use the results of the focused experiments to drive further scanning, which leads to new questions and additional experimentation. REs iterate between subcomponent scanning and focused experimentation, learning more about the program until they can answer their target questions (e.g., identify a vulnerability, understand a malware infection).

Based on this model and our interviews, we established the following five usability guidelines:

- G1 **Match interaction with analysis phases.** Reverse engineering tools should be designed to facilitate each analysis phase: overview, subcomponent scanning, and focused experimentation.
- G2 **Present input and output in the context of code.** Integrate analysis interaction into the disassembler or decompiled code view to support tool adoption.
- G3 **Allow data transfer between static and dynamic contexts.** Static and dynamic analyses should be tightly coupled so that users can switch between them during exploration.
- G4 **Allow selection of analysis methods.** When multiple options for analysis or levels of approximation are available, ask the user to decide which to use.
- G5 **Support readability improvements.** Infer semantic information from the code where possible and allow users to change variable names, add notes, and correct decompilation to improve readability.

In this paper, we utilize the three-phase model from our prior work to ground our analysis of tool purpose (Section 4.3) as it presents the most thorough representation of the reverse engineering process. This model has also been validated by subsequent large-scale RE studies [20, 40] and similar qualitative RE investigations, e.g., into dynamic malware analysis [66] and hardware reverse engineering [9]. We also found the three phases were sufficient to accurately apply labels to all tools.

3 RELATED WORK

While we believe our work is the first to thoroughly investigate current reverse engineering tool usability, there has been significant prior effort to survey current reverse engineering tools' functionality, and develop more usable tooling on a technique-by-technique basis. In this section, we summarize and compare our study to this prior work.

3.1 Tool Surveys

Perhaps most similar to our survey is the twice-annual Edge of the Art report by TwoSix Labs, which outlines advancements in cyber security community tools, focusing on reverse engineering tools [59]. This living document captures available tools, their capabilities, program analysis theories and concrete approaches employed, as well as a general notion of tool usability. However, they do not explicitly define or cite an assessed definition of usability. We establish usability criteria as a primary assessor of various reverse engineering tools and conduct a more in-depth and rigorous qualitative coding process.

Several prior papers have reviewed existing reverse engineering tools as part of systemizations of the program analysis literature in various sub-areas. For example, Schwartz et al. reviewed dynamic taint analysis and forward symbolic execution tools to produce a precise definition of these techniques [53]. Arusoaie et al. review open-source static analysis C/C++ vulnerability discovery tools to compare their detection and false positive rates [5]. Similarly, Shoshitaishvili et al. survey and reproduce offensive binary analysis tools into a single coherent framework to allow comparison [55]. This prior work outlines the various functionalities and presents existing tools, but they do not consider usability in their review. We use these reviews, along with the Edge of the Art report as seeds for our tool search (Section 4.1) and to guide our functionality definitions (Section 4.3.3).

3.2 Technique-Specific Usability

Prior work has also focused on usability challenges specific to various program analysis techniques or aspects of reverse engineering. For example, Ploger et al. studied the usability challenges beginners faced when trying to setup and run a fuzzer and static analyzer, showing that the challenge of setting up these tools posed a significant barrier to use [48]. Focusing on the issue of seed generation in the context of mutational fuzzers, Shoshitaishvili et al. developed the HaCRS system, which created an interface allowing general users to provide input seeds without any fuzzer or even programming knowledge [56].

Another area of focus has been on improving program readability. Perhaps the best example is Yakdan et al. development of DREAM++, a usability-optimized decompiler, which used heuristic-based transformations to improve decompiled code readability. Similar tools have been developed using various AI/ML techniques to infer variable/function names, object structures, and types, to aid program comprehension [3, 4, 11, 35, 37].

We have also previously considered reverse engineering plugins, but focused specifically on the NSA's Ghidra framework [61]. Through a review of 1590 community forum posts directly before and after the public release of Ghidra, we sought to understand the functionality and usability concerns REs have about Ghidra by analyzing the questions asked. We observed the ability to customize tools was of primary importance to REs, but were unable to show specific aspects of customization, nor provide significant insights into tool usability questions due to the lack of comparison with other tools. This work investigating specific aspects of usability and developing more usable tools is important because it leads to better tooling. However, we chose to take a broader, more holistic view of the reverse engineering tools landscape to understand the breadth of usability issues and guide future, similar, tailored work.

4 METHODS

In this section, we discuss how we identified RE tools for evaluation, our qualitative coding process, and our quantitative analysis of the coded results. An overview of our workflow is given in Figure 1.

4.1 Tool Search Process

Our first step was to identify current reverse engineering tools (Figure 1.A). RE tools can be divided into two categories: standalone and framework plugins. Standalone operate as a self-contained program, whereas, plugins function as an extension of a broader tooling framework.

Framework	Tools Found	Tools Coded
Ghidra	75	56
IDA	204	101*
Binary Ninja	78	47
Radare2	50	34
Standalone	55	50
Total	**462**	**288**

Table 1: Total number of tools found through our search and the number qualitatively coded after applying our exclusion criteria. The * indicates that the count of IDA plugins coded was a random sample of 62% of the 162 tools after initial exclusion review.

REs have a choice between several security frameworks that offer similar baseline functionality. These frameworks provide binary disassembly and sometimes decompilation, variable name editing, and other functionalities. Each framework allows REs to create plugins to extend the features, leveraging the information and UIs produced by the framework. Based on discussions with seven RE experts and prior literature [55, 59, 61–63], we identified four popular, extendable RE frameworks to use for our review: Ghidra, IDA, Radare2, and Binary Ninja.

To ensure an extensive tool survey, we applied multiple search methods. We performed keyword searches of Github, Google, and Twitter. We also reviewed each framework's web pages looking for downloadable plugins to run and considered tools mentioned in prior literature reviews [5, 53, 55, 59]. The specifics of our tool search process are given in the supplemental material [2].

Our RE tool search was conducted between January and May of 2021. We identified 462 unique tools (407 plugins; 55 standalone). The final counts of identified tools are given in the second column of Table 1.

4.2 Exclusion Criteria

We cast a broad net for reverse engineering tools. However, not all plugins or standalone tools provide functionality relevant to our review. After the initial search, we refined our list using the exclusion criteria below.

RE-specific tools only. We were concerned with tools that provide functionality specifically for the reverse engineering process. Tools with a focus on utility were excluded, such as Screenshot Ninja which provides a more streamlined way to take screenshots in Binary Ninja. 53 non-RE-specific tools were excluded from review.

Extending architecture support. We excluded plugins that added support for different instruction set architectures to a framework without including any new functionality. For example, plugins for loading popular gaming systems, such as Sega Genesis [42] and Gameboy [49], which allow REs to analyze games produced for these systems. We excluded 65 architecture support plugins.

Random sample of IDA plugins. We coded every remaining Ghidra, Binary Ninja, and Radare2 plugin and standalone tool after

[2] https://osf.io/ec7j9/

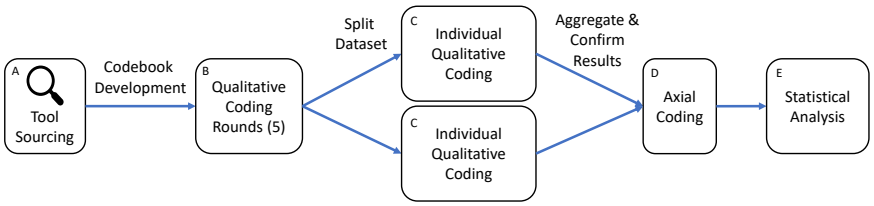

Figure 1: Workflow diagram showing our methodology from sourcing tools to final statistical analysis

	Variable	Definition	Values	α
RQ1	Input Content	The types of information provided as tool input	See Table 4	1
	Output Content	The types of information provided as tool output	See Table 5	.89
	Output Method	How the output is presented	See Table 6	0.843
RQ2	Analysis Phase	Which RE mental model analysis phase the tool is used in	Overview/Subcomponent/ Experimentation	.815
	UI in Code	The tool presents input and output in the code viewer	Yes/No	–*
	Static+Dynamic	The tool operates on static and dynamic information	Yes/No	1
	Analysis Tuning	The tool allows the RE to tune the analysis	Yes/No	1
	Readability	The tool improves the program's readability	Yes/No	1
	Functionality	What static and/or dynamic analysis does the tool perform	See Tables 7 and 8	0.833

Table 2: An overview of the codebooks used in our reverse engineering tool qualitative coding process. Detailed codebook descriptions are provided in Appendix B. Note, adherence to the *UI in Code* guideline was determined based on the RQ1 coding results and thus no additional IRR calculation was required.

applying our exclusion criteria. However, IDA had more than double every other framework. Because of the significant amount of time necessary to analyze each plugin, we chose to perform a random sample of the IDA plugins. Specifically, we qualitatively coded tools until we reached thematic saturation, i.e., no new themes were identified with further review—a common stopping criteria for initial coding—and we had coded at least 50% of the plugins (i.e., 100 plugins). However, because some plugins were removed according to our exclusion criteria, the final sample was at least 62%[3] of the total IDA plugins. We coded 288 tools (238 plugins; 50 standalone).The final number of tools coded for each framework and standalone tools are given in Table 1.

4.3 Qualitative Analysis

To begin analyzing our sample of reverse engineering tools, we performed a qualitative coding of each tool considering multiple variables related to interaction modalities, tool usability, and tool functionality. In this section, we describe each coded element, as well as our coding process. Table 2 provides a summary of our codebook.

In qualitative coding, the researchers' expertise impacts the codebook and code application, especially with domain-specific topics. Our team included one researcher with eight years of professional RE experience, and tool development and usability testing experience; one researcher with one year of professional RE experience, and one researcher with limited prior RE experience. The two less experienced researchers coded the tools and consulted the expert to ensure appropriate code use. The codebook was designed so codes were straightforward and objective with positive and negative examples.

[3]This percentage may be higher as some unreviewed plugins would have been removed based on our exclusion criteria after review.

4.3.1 Interaction Modalities (RQ1). To evaluate how REs interact with their tools, we analyzed the input and output options. Specifically, we coded three variables: input content, output content, and output method.

Input Content. First, we consider what types of input an RE can provide the tools. This includes direct inputs (e.g., input file, command line flags, GUI) and configuration information (e.g., configuration files, environment flags).

Output Content. Next, we reviewed what types of information are given as output. This could be provided at the end of the tool's execution or intermittently as the tool operates and the user interacts with it.

Output Method. For the output, we also considered how and where this information was presented. This includes output on the command line, in a GUI, as well as different ways outputs could be visualized. Note, we did not consider input method because the way in which the RE provides the input is often inherent in the type provided.

4.3.2 Usability Evaluation (RQ2). To evaluate how existing plugins align with the reverse engineering process and mental models, we identified which phase(s) of the three-phase RE process model [62] it was intended to be used during, and coded whether the tool applied each of our previously established usability guidelines. We discuss the coding of phases and guidelines below.

Match interaction with analysis phase (*Analysis Phase*; G1). As we discussed in Section 2, our prior work identified three phases of the reverse engineering process: overview, subcomponent scanning, and focused experimentation. To understand where each tool fits with the reverse engineering process, we included a binary variable for each phase indicating whether the tool could be used in that phase based on its functionality. We found the three phases to be sufficient when applying labels to each tool. Every tool fit

into one (or more) of the three phases and a tool was never forced into a phase.

Our first proposed usability guideline suggests tools should support transitions between each phase. When tools had functionality covering multiple phases, we evaluated whether the functionality in each phase supported transitions between phases. If the tool was coded for two neighboring phases (i.e. overview/subcomponent scanning or subcomponent scanning/focused experimentation) and the information from the plugin could be used across both phases, the tool would satisfy G1.

Present input and output in the context of code (*UI in Code*; G2). This usability guideline was not explicitly coded in our codebook, but was instead captured through the different input and output modalities we measured for each tool. Whenever a tool would receive input or display output directly integrated with the code viewer or code execution graphs, the tool would also be marked as satisfying G2. Any tools that directly modify the original binary or annotate the program also fall under G2.

Allow data transfer between static and dynamic contexts (*Static+Dyanmic*; G3)). Our prior work suggested reverse engineering tools support the transition between static and dynamic contexts. This support reduces the cognitive load on REs as they switch between static and dynamic methods. For each tool, we considered whether it helped users reason over static and dynamic information by mixing these two types of analyses internally or providing output from one context in a presentation of the other. For example, if a tool presented the outputs of a dynamic execution trace within a disassembled code view, it would be marked as meeting this guideline.

Allow selection of analysis methods (*Analysis Tuning*; G4). We evaluated whether reverse engineering tools allow the RE to examine results with varying levels of approximation. Throughout the RE process, REs choose different methods to use based off their prior experience and previous results. By enabling users to adjust how a tool operates to better fit their needs, tools allow REs to better explore their hypotheses and validate their results.

Support readability improvements (*Readability*; G5). Improving program readability is a key step in the reverse engineering process as REs try to better simplify and understand the inspected code. Many REs spend time manually annotating different variables or function signatures with semantic information about their use and functionality. Tools that automate this process can expedite reverse engineering. We include a binary variable in our analysis to indicate whether the tool produces output to make it easier for an RE to understand a program.

4.3.3 Functionality Type. In addition to understanding the usability of reverse engineering tools generally, we also sought to identify differences between different tool functionalities. Functionality types were divided between static and dynamic functions and we began with definitions from prior literature reviews in program analysis and tool surveys [5, 53, 55, 59]. Multiple functionalities (static and dynamic) could be coded for a single tool.

4.3.4 Analysis process. Two research team members developed the initial codebook by cooperatively coding the 47 Binary Ninja plugins using iterative open coding [17], and through discussions with the full research team (Figure 1.B). Next, two researchers independently coded tools in groups of 10 (switching to groups of 20 after 3 rounds) beginning with the initial codebook and allowing additional codes to emerge from the data. After each round, the researchers met to compare codes, resolve disagreements, update the codebooks, and when necessary, re-code tools(Figure 1.C). We calculated Krippendorff's alpha (α) using the ReCal2 software package [21] to measure inter-coder reliability. We used Krippendorff's alpha as it provides a conservative measure, accounting for chance agreements [30]. This process was repeated five times until α exceeded 0.80 for each variable, Krippendorff's recommended threshold [30]. Final α values for each variable are reported in Table 2. The remaining tools were divided between the two researchers and coded independently by a single researcher.

After completing our open coding, we performed axial coding (Figure 1.D) to determine groups of codes for each variable [58, pg. 123-142]. Axial coding identifies connections between codes to extract higher-level representations that reduce the number of comparisons for our quantitative analysis (described next, in Section 4.4).

To evaluate each tool on our codebook criteria first we reviewed the provided documentation. Many of the tools have extensive documentation with sample images and videos that are detailed enough to fully fill out our codebook. In the case where the documentation was insufficient, we download and install the tool as per the installation instructions. Several tools provided toy binaries to test the functionality of the tool which could be repeatedly used for other tools of similar functionality.

4.4 Quantitative Analysis

With the resulting data, we sought to compare trends in the frequency of different interactions and the likelihood of each usability guideline being adopted. Because tool design varies based on functionality and purpose, we consider the functionality type and analysis phase in our analysis. Since the four frameworks support the entire RE process, they offer similar features with similar plugin types. Our sample size limited the number of statistical comparisons we had sufficient power to perform, therefore we did not include inter-framework comparisons.

Interaction modality comparisons (RQ1). To compare trends in interaction modalities, we performed a series of Chi-Squared tests—appropriate for categorical data [22]—to compare the tools including each interaction modality type against those with another, assessing the effect size (ϕ) and significance ($p - value$) of the difference. For each interaction modality variable (i.e., input content, output content, and output method), we performed three sets of tests each with a different covariate: one comparing modalities in each of the three analysis phases, one comparing modalities in tools implementing static versus dynamic functionality, and one comparing modalities in integrated plugins, standalone tools, and client plugins. For each set of comparisons, we began with an omnibus test to determine whether any statistically significant difference existed between modalities across the covariates. If the omnibus produced a statistically significant result, we performed a full-factorial pairwise comparison between the modality's groups and the tested

covariate. Because we are doing multiple comparisons, we adjust the results using a Benjamini-Hochberg (BH) correction [10].

Usability guideline comparisons (RQ2). For each guideline, we performed a logistic regression to understand whether the analysis phase or functionality type were correlated with guideline adherence. In each initial model, our independent variables included a binary variable for each analysis phase (overview, subcomponent scanning, and experimentation) indicating whether the tool was used in that phase, as well as a functionality type variable indicating whether the tool provided static, dynamic, or both functionalities. The dependent variable for each model was a binary variable indicating whether tool applied the relevant guideline. We calculated the Bayesian Information Criterion (BIC)—a standard metric for model fit [63]—for all possible combinations of the initial independent variables. To determine the optimal model and avoid over-fitting, we selected the minimum BIC model.

4.5 Limitations

There are several limitations inherent to our methodology. First, we evaluate tools based on a heuristic model of usability, not through a user study. It is likely there are tool design elements which make the reviewed reverse engineering tools more or less usable, not captured in our heuristics. However, it is not feasible to recruit a sufficiently large sample of REs to conduct a broad interactive usability review of reverse engineering tools. Therefore, our work offers a necessary first pass to provide focus for future user studies. We were not able to find any reliable metric to gauge tool popularity. Knowing tool popularity would have allowed us to weigh our analysis toward tools that are more commonly used and therefore effect more REs. Initially we considered using Github's Stargazers, but we found this feature was not used often by REs and is not a reliable measure of popularity. Further, not all reverse engineering tools are distributed on Github, so this was not comparable across the sample. Although we encountered many duplicate listings of the same tool, a selection of unique tools will not be captured in our data set. However, due to our extensive search process, we believe our sample represents a large enough portion of the total population to uncover representative trends.

5 TOOL CHARACTERISTICS

We begin by setting the stage for our analysis by discussing the 288 coded reverse engineering tools' general characteristics. We report the coded tools' functionality and phase of use to demonstrate how and when they are used. These results characterize our sample and operate as covariates for our primary analyses (Sections 6 and 7).

5.1 Functionality

First, we describe the types of tool functionality divided into static and dynamic functionality. For brevity, we do not provide detailed descriptions of each functionality type and only describe representative tool examples and trends. The full list, taken from prior work [5, 53, 55, 59], and functionality counts are given in Appendix B.

Static analysis was most common. Most tools included static analyses (85%, 244/288). This was dominated by heuristic scanning (61%, 148/244), which search the file for a pattern (often a string) and highlights instances. Inter-tool transfer (20%, 49/244), which allowed REs to import/export data to allow data sharing between tools, was the second most common. Several tools produced code visualizations (17%, 42/244), such as control flow or code complexity graphs. Another common static type was tools that modified a framework's function (12%, 29/244), such as Ghidra-Jupyter which creates a Jupyter notebook linked to the RE's Ghirda instance [24], but do not provide functionality for a specific reverse engineering task. Other common static analyses include symbolic execution (7%, 18/244) and disassemblers (7%, 17/244).

Dynamic analysis was most common in standalone tools. Dynamic functionality was less common overall (29%, 84/288), but it was much more common in standalone tools—68% had dynamic functionality (34/50). There were only a few common dynamic analysis types: fuzzers (33%, 28/84), debuggers (30%, 25/84), and tools that patch a binary to allow dynamic logging (27%, 24/84). These three types made up 88% of dynamic analyses.

5.2 Reverse Engineering Phase

Next, we describe the reverse engineering phases in which tools were designed to be used.

Framework plugins were most often designed for the overview phase. 68% of plugins (163/238) were designed to provide REs a high-level overview of the program. Further, most plugins were designed only to provide an overview (59%, 141/238). Conversely, few standalone tools were used in the overview phase (42%, 21/50).

Focused experimentation was most common in standalone tools. Standalone tools primarily focused on supporting REs as they test hypotheses and ask specific questions. 64% of standalone tools (32/50) were designed to be used during focused experimentation and 58% (29/50) exclusively so. Whereas, 15% of plugins (36/238) were used in only the focused experimentation phase.

Subcomponent scanning was the least common analysis phase. Tools meant to be used in the subcomponent scanning phase were the least common, with 18% (53/288) having some subcomponent scanning functionality and only 11% (33/288) used exclusively in this phase.

6 INTERACTION MODALITIES (RQ1)

Next, we describe the different interaction modalities available to REs to interact with the tools divided by input content, output content and output method. The full list and counts of different input/output types are given in the Appendix in Tables 4, 5, and 6.

6.1 Input Types

Many plugins only take a file as input. The vast majority of reverse engineering tools (91%, 261/288) take a binary or assembly file as input as the analysis target. However, in 57% of tools (164/288), this is the only input. This lack of interaction is more common in plugins where 62% (148/238) only take an input file versus 32% of standalone tools (16/50). For example, many heuristic scanning plugins include hard-coded search patterns, requiring the RE to edit plugin code to make adjustments.

Selected area is the most common input. Several tools allow an

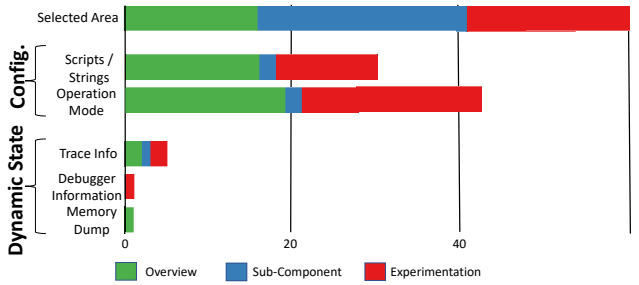

Figure 2: Stacked bar chart showing breakdown of analysis phase by Input content type. The different groupings shown are used for Chi-squared tests. Binary file and Assembly file inputs are not shown as those two entries are present in 261 different tools.

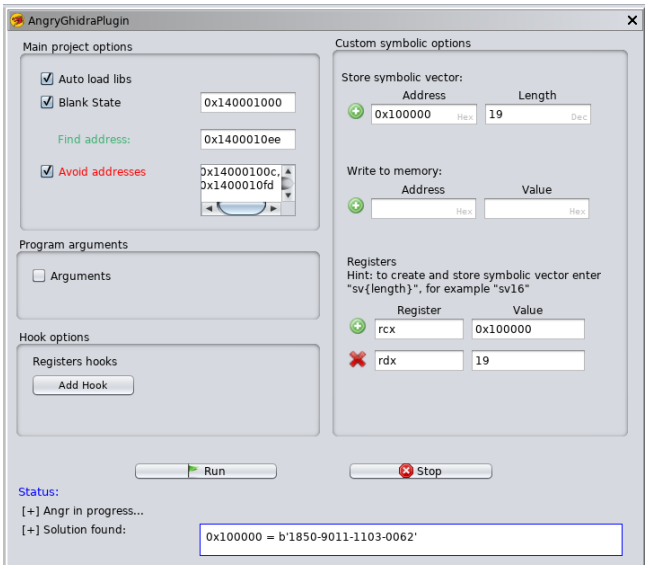

Figure 3: Screenshot of Angr plugin for Ghidra showing possible RE inputs [43].

RE to select an area of the program, e.g., class, function, or segment of code, to focus the analysis (17%, 49/288). For example, Figure 3 shows Angry Ghidra, a plugin that allows REs to indicate target addresses to avoid during symbolic execution. These inputs are the most commonly observed, as shown in Figure 2, which gives the number of tools providing each input type.

Some tools allow analysis configuration. Some tools allow for Analysis configuration. REs can provide user-defined scripts and string lists (8%, 24/288), as well as operation mode flags (15%, 42/288). In the first case, REs provide custom scripts or strings that direct tool function. For example, climacros allows REs to use macros in IDA's command-line interface [7] to run user created scripts.

Standalone tools were more likely to allow operation mode configuration. Tools were coded as providing operation mode when command-line flags or configuration files could be modified to change the analysis functionality. This input type was more common among standalone tools (44%, 22/50) than plugins (8%, 20/238). For example, many fuzzers allow REs to adjust parameters using command-line flags. Standalone tools make significantly more use of configuration information compared to selected area than plugins (χ^2 = 8.65, p = 0.013).

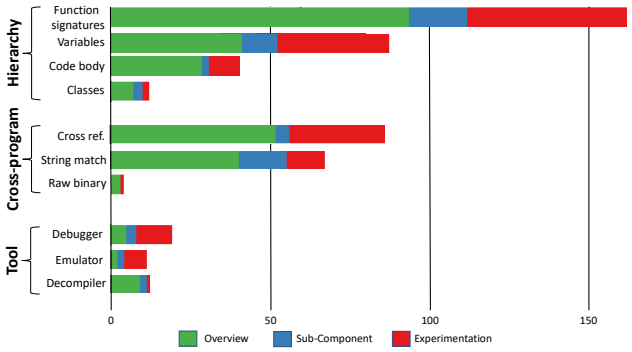

Figure 4: Stacked bar chart showing breakdown of analysis phase by Output content type. The different groupings shown are used for Chi-squared tests.

Dynamic state information provided by another tool was least common. Only 2% (7/288) of tools utilized dynamic state information from another tool or between tool functionalities. This includes four tools that accept program traces to highlight reached code segments in the code viewer and a plugin that links gdb with the IDA console to show the current line of code being debugged [25]. We specifically consider dynamic state information provided from tool-to-tool or function-to-function, as opposed to dynamic state generally. All tools with dynamic functionality operate on dynamic state inherently as all static tools operate on the target program code.

Selected area input is more common in subcomponent scanning. Selected area input was significantly more common than configuration input in the subcomponent scanning phase than the overview (χ^2 = 12.72, p = 0.002) and focused experimentation (χ^2 =15.73 , $p < 0.001$) phases.This is expected as both subcomponent scanning and focused experimentation phases have a primary focus on a program subset. Subcomponent scanning tools differ from focused experimentation tools in their lack of other interaction options. While 18% (33/184) of overview and 43% (36/83) of focused experimentation tools provide configuration and dynamic state inputs, this drops to 11% (6/53) for subcomponent scanning tools. We did not observe any other statistically significant differences between inputs across analysis phases or functionality types. A full listing of our statistical test results after correction are given in the supplemental material.

6.2 Output Types

Output most often highlighted program structure. A majority of reverse engineering tools presented their output in the context of the program's hierarchical elements (73%, 209/288). This included modifying or displaying function names and signatures (52%, 151/288), variable names (27%, 79/288), classes (4%, 11/288), and other statements throughout the program body (9%, 26/288).

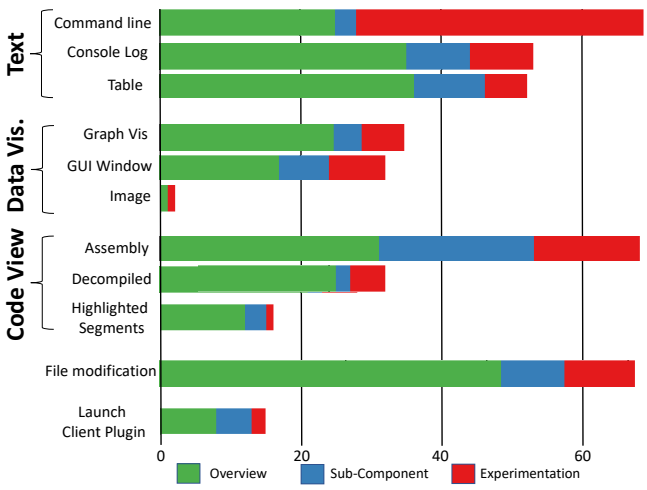

Figure 5: Stacked bar chart showing breakdown of analysis phase by Output method type. The different groupings shown are used for Chi-squared tests.

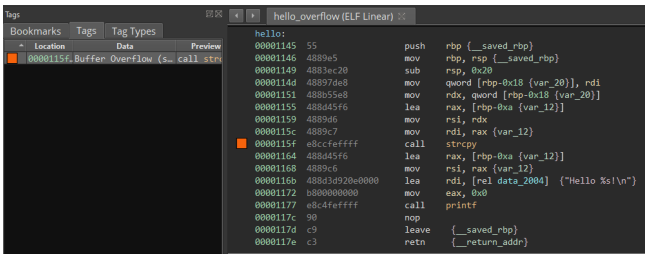

Figure 6: Screenshot of VulnFanatic which highlights vulnerable function calls [41].

Several plugins made use of multiple structural elements (37%, 107/288) like IFL which creates an index-able table of functions, arguments, and cross references the RE can use to navigate and locate areas of interest [28].

Many tools highlighted cross-program relationships. Several tools produced output regarding relationships across the program, allowing the RE to identify links between components or focus on segments with particular characteristics (45%, 129/288). This included listing a function's cross-references (26%, 75/288) or showing all locations where a particular string or regular expression matched (22%, 62/288). For example, FindCrypt can scan a binary for common crypto and hashing constants to help the RE identify which algorithms are used [52]. Finally, some tools highlighted patterns in the raw binary file (1%, 3/288), such as high entropy sections, which might suggest encrypted or packed code [39].

Dynamic tools are more likely to produce or extend supplementary tool output. 28% (25/88) of dynamic tools produce output making use of tool information. This includes debugging information (16% 14/88), emulator information (11% 10/88), and decompiler information (1% 1/88). This differs significantly from static tools where only 11% (28/244) produced tool information. Our Chi-squared tests showed static tools are significantly more likely to make use of hierarchical (χ^2 = 15.05, $p < 0.001$) and cross-program (χ^2 = 25.27, $p < 0.001$) information compared to dynamic tools. As previously mentioned, most focused experimentation plugins are dynamic tools. Therefore it follows that focused experimentation tools are significantly more likely to make use of tool information than cross-program information compared to overview tools (χ^2 = 6.72, p = 0.033).

Output was most often reported in text. Despite most reverse engineering tools being integrated plugins that could leverage a framework GUI, the most common method for results display was plain text (49%, 116/238). This includes information displayed on the integrated console log (20%, 48/238), an external command line (10%, 25/238), and tables of text (18%, 42/238). Standalone tools are particularly limited in their output methods, with only seven using a visualization or GUI (15%, 7/50).

Some tools presented output in a code viewer. Despite 83% of our data consisting of plugins, only 31% (89/288) of tools used the code viewer to display results. This included modifying the assembly (24%, 58/238) or decompiled code viewer (12%, 31/238), or highlighting relevant code segments (7%, 16/238) as shown by VulnFanatic in Figure 6. Static tools are significantly more likely to use the code viewer than text (χ^2 = 12.60, p = 0.006). Plugins were significantly more likely to use the code viewer than text (χ^2 = 14.96, p = 0.002) and file modifications (χ^2 = 6.97, p = 0.046), compared to standalone tools

Fewer tools utilized data visualizations. It was even more uncommon for reverse engineering tools to produce a graphical visualization of the program to allow quick review (10%, 20/288) or create their own pop-up or GUI window (10%, 28/288). This highlights the fact that reverse engineering tool developers are unlikely to create their own visualizations in current frameworks.

A few tools produced runnable output. Some tools created executable output files as the analysis product (19%, 54/288). This included modifying the target binary (8%, 23/288), such as Keypatch, which allows REs to insert assembly instructions [19]. Other tools produced scripts based on the analysis that could be applied in later analysis (4%, 12/288). For example, IDA-climacros allows REs to create static and dynamic macros to invoke during scripting or while navigating the IDA GUI [7].

7 USABILITY GUIDELINES (RQ2)

Next, we discuss how current tools adhere to previously suggested usability guidelines [62].

Supporting transitions between analysis phases was rare (G1). While some tools included functionality in multiple phases (11%, 33/288), few supported transitions between phases (7%, 20/288). For example, a heuristic scanning tool could identify potentially vulnerable functions, but there was no tool interaction provided to pass these results to a dynamic analysis for follow-on review.

One tool that helps facilitate the transition between phases is Snippet Detector [67]. Snippet Detector allows a RE to add reverse engineered code segments (i.e., snippets) to a database and label them with relevant semantic information (e.g., rename functions), a common process during subcomponent scanning. Then, the tool

Variable	Value	OR	CI	p-value
Subcomponent	F	-	-	-
	T	**5.95**	**[2.78, 12.74]**	**0.001**

(a) UI in Code (G2)

Variable	Value	OR	CI	p-value
Overview	F	-	-	-
	T	3.07	[0.97, 9.65]	0.055
Subcomponent	F	-	-	-
	T	2.45	[0.71, 8.42]	0.15
Functionality	S	-	-	-
	D	**8.05**	**[1.95, 33.14]**	**0.004***
	S/D	**173.32**	**[49.65, 605.01]**	**<0.001***

(b) Static+Dynamic (G3)

Variable	Value	OR	CI	p-value
Experimentation	F	-	-	-
	T	**4.03**	**[2.32, 6.99]**	**<0.001***

(c) Analysis Tuning (G4)

Variable	Value	OR	CI	p-value
Experimentation	F	-	-	-
	T	0.59	[0.29, 1.21]	0.15
Functionality	S	-	-	-
	D	**0.09**	**[0.02, 0.33]**	**<0.001***
	S/D	**0.39**	**[0.18, 0.86]**	**0.019***

(d) Readability (G5)

Table 3: Results from logistic regression for usability guidelines G2-5 vs analysis phase and functionality type. Odds ratios greater than one imply the plugin is more likely to meet the usability criteria than the base case, and less than one imply the plugin is less likely to achieve the guideline than base case. Significant effects are indicated with a *. Under functionality, D indicates dynamic, S is static.

identifies syntactically or semantically matching snippets, providing overview support, and applying the RE-defined labels, simplifying later subcomponent scanning. We did not observe statistically significant differences between phases or functionality types; the final regression results can be found in the supplemental material.

The majority of plugins used input/output in the context of code (G2). 58% (139/238) of plugins allowed the RE to interact with the tool in the context of code. This is most common in plugins which present the results in the framework's code viewer (62% 86/139). Only 22% (11/50) of standalone tools met this guideline. Tools were also more likely to present input and output in context of code in the subcomponent phase (OR = 5.95, $p < 0.001$).

Tools that combine static and dynamic contexts are predominantly experimentation and dynamic. While few tools integrated static and dynamic contexts overall (17%, 48/288), 52% (25/48) are used in the experimentation phase and 83% (40/48) included dynamic functionality. Many were fuzzers leveraging symbolic execution to expand their fuzzing path (20%, 10/48). For example, Driller which combine symbolic execution with fuzzing to allow the fuzzer to reach deeper portions of the program [57]. 44% (21/48) of tools that combined static and dynamic data also allowed analysis tuning. This suggests the more advanced tools often give REs more flexibility.

Analysis tuning was most common during focused experimentation (G4). Few tools allowed REs to tune their analysis iteratively (28%, 81/288). Only 22% (41/184) overview tools and 23% (12/53) subcomponent scanning tools allow analysis tuning, however this distribution flips with focused experimentation (49%, 41/83). As shown in Table 3c, if a plugin is used during focused experimentation it is over 4 times ($p < 0.001$) more likely to allow analysis method selection. This result was dominated by the use of debuggers during focused experimentation, which are inherently iterative, allowing the user to step through program execution and introspect memory.

Readability improvements are most common (G5). Many tools aimed to improve readability (46%, 133/288). 80% (106/133) of tools that improved readability are static scanning tools that highlight areas of code, or replace different matched strings. Readability improvements were also statistically significantly more likely when tools employed static functionality types than tools with just dynamic functionality (OR = 0.09, $p < 0.001$) or both static and dynamic functionality (OR = 0.39, $p = 0.019$).

8 DISCUSSION

Our results indicate reverse engineering tools' usability is generally low. Most do not support input beyond the target file and many do not provide output in the context of code or in a form beyond raw text. While most tools adhere to at least one (78%, 226/288) or two (49%, 141/288) usability guidelines, few meet more than half (15%, 43/288). However, the reason for limited usability is not uniform across tool functionality. Specifically, we observed the following overarching trends:

- Dynamic tools were less likely to improve readability and ease of interaction. Static tools were often integrated into an reverse engineering framework, leveraging existing UIs to provide varied interactions and improve readability.
- Static tools allow less configuration and analysis tuning. To modify program analysis, REs must modify the tool itself. Conversely, Dynamic tools allowed REs to adjust analysis parameters through configuration options.

Based on these results, we suggest the following recommendations to improve reverse engineering tool usability.

Improve reverse engineering framework interaction support. The lack of interaction in standalone tools is likely caused by the challenge of building new UIs for each tool. Our results and prior work show reverse engineering tool developers are capable of producing inventive new analyses and functionalities. However, interface development requires additional time and different skills. Instead, reverse engineering tool developers rely on the API support from the frameworks, rarely opting to create their own UIs (10%, 28/288). We believe frameworks can play a large role in improving reverse engineering tool usability by adding more interaction support. Reverse engineering frameworks already provide support for static functionalities, which led to those tools providing richer interactions and meeting the *UI in Code* and *Readability* guidelines. However, there is limited support for dynamic analysis, which is likely why these tools dominate the standalone category. To improve usability, reverse engineering frameworks should seek to

improve APIs for UI integration of dynamic analyses, and support for combining static and dynamic analysis results.

HCI research in reverse engineering tool usability. We observed several areas where available modes of interaction were limited. There was very little interaction in the static tools for analysis tuning beyond selecting a particular area. Even in the dynamic setting, the tuning options were often limited to text-based configuration files and command-line options, not interactively throughout use. Additionally, we observed few available interactions in the subcomponent scanning phase. These gaps present a clear opportunity for future research in interaction design. Because there are limited options in this space, this suggests the need for exploratory work to investigate user needs and patterns of manual interaction throughout their tool use and modification of the tool's code. Based on use-inspired investigations, previous approaches from other exploratory analysis domains, such as exploratory visual analysis [8, 31, 36, 47, 54, 65], may be appropriate or novel interactions may be necessary to allow REs to run, review, and adjust the analysis accordingly.

ACKNOWLEDGMENTS

We thank the reviewers who provided helpful comments and Jordan Wiens and Rock Stevens for valuable insights on our study design. The author's affiliation with The MITRE Corporation is provided for identification purposes only, and is not intended to convey or imply MITRE's concurrence with, or support for, the positions, opinions, or viewpoints expressed by the author. MITRE has approved this work for public release and unlimited distribution; public release case number 22-3565.

REFERENCES

[1] US National Security Agency. 2021. Ghidra. https://github.com/NationalSecurityAgency/ghidra (Accessed 08-11-2021).
[2] Omer Akgul, Taha Eghtesad, Amit Elazari, Omprakash Gnawali, Jens Grossklags, Daniel Votipka, and Aron Laszka. 2020. The Hackers' Viewpoint: Exploring Challenges and Benefits of Bug-Bounty Programs. In *2020 Workshop on Security Information Workers (WSIW '20)*. USENIX Association, Virtual. https://wsiw2020.sec.uni-hannover.de/downloads/WSIW2020-The%20Hackers%20Viewpoint.pdf
[3] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting Accurate Method and Class Names. In *2015 Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE '15)*. Association for Computing Machinery, New York, NY, USA, 38–49. https://doi.org/10.1145/2786805.2786849
[4] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.* 3, POPL, Article 40 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290353
[5] Andrei Arusoaie, Stefan Ciobâca, Vlad Craciun, Dragos Gavrilut, and Dorel Lucanu. 2017. A Comparison of Open-Source Static Analysis Tools for Vulnerability Detection in C/C++ Code. In *2017 International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC '17)*. 161–168. https://doi.org/10.1109/SYNASC.2017.00035
[6] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. 2014. Automatic Exploit Generation. *Commun. ACM* 57, 2 (feb 2014), 74–84. https://doi.org/10.1145/2560217.2560219
[7] Elias (0xeb) Bachaalany. 2021. IDA Command Line Interface Macros. Accessed: 2022.
[8] Leilani Battle and Jeffrey Heer. 2019. Characterizing exploratory visual analysis: A literature review and evaluation of analytic provenance in tableau. In *Computer graphics forum*, Vol. 38. Wiley Online Library, 145–159.
[9] Steffen Becker, Carina Wiesen, Nils Albartus, Nikol Rummel, and Christof Paar. 2020. An Exploratory Study of Hardware Reverse Engineering — Technical and Cognitive Processes. In *Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020)*. USENIX Association, 285–300. https://www.usenix.org/conference/soups2020/presentation/becker
[10] Yoav Benjamini and Yosef Hochberg. 1995. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society. Series B (Methodological)* 57, 1 (1995), 289–300. https://doi.org/10.2307/2346101
[11] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. 2016. Statistical Deobfuscation of Android Applications. In *2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 343–355. https://doi.org/10.1145/2976749.2978422
[12] Adam Bryant. 2012. *Understanding How Reverse Engineers Make Sense of Programs from Assembly Language Representations*. Ph. D. Dissertation. US Air Force Institute of Technology.
[13] Mariano Ceccato, Paolo Tonella, Cataldo Basile, Bart Coppens, Bjorn De Sutter, Paolo Falcarin, and Marco Torchiano. 2017. How Professional Hackers Understand Protected Code While Performing Attack Tasks. In *2017 International Conference on Program Comprehension* (Buenos Aires, Argentina) *(ICPC '17)*. IEEE Press, Piscataway, NJ, USA, 154–164. https://doi.org/10.1109/ICPC.2017.2
[14] Mariano Ceccato, Paolo Tonella, Cataldo Basile, Paolo Falcarin, Marco Torchiano, Bart Coppens, and Bjorn De Sutter. 2019. Understanding the behaviour of hackers while performing attack tasks in a professional setting and in a public challenge. *Empirical Software Engineering* 24 (2019), 240–286. Issue 1. https://doi.org/10.1007/s10664-018-9625-6
[15] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *2012 IEEE Symposium on Security and Privacy (SP '12)*. IEEE Computer Society, Washington, DC, USA, 380–394. http://dx.doi.org/10.1109/SP.2012.31
[16] Kathy Charmaz. 2006. *Constructing Grounded Theory: A Practical Guide Through Qualitative Analysis*. SagePublication Ltd, London. http://www.amazon.com/Constructing-Grounded-Theory-Qualitative-Introducing/dp/0761973532
[17] Juliet Corbin and Anselm Strauss. 2014. *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Sage publications.
[18] Eldad Eilam. 2011. *Reversing: secrets of reverse engineering*. John Wiley & Sons.
[19] Keystone Engine. 2021. KeypatchKeystone Engine. Accessed: 2022.
[20] Kevin Burk Fabio Pagani, Christopher Kruegel and Giovanni Vigna. 2022. How Humans Decompile and What We Can Learn From It. *Proceedings of the 31st USENIX Security Symposium* (2022), 2765–2782.
[21] Deen G Freelon. 2010. ReCal: Intercoder reliability calculation as a web service. *International Journal of Internet Science* 5, 1 (2010), 20–33.
[22] Karl Pearson F.R.S. 1900. On the Criterion that a Given System of Deviations from the Probable in the Case of a Correlated System of Variables is Such that it Can be Reasonably Supposed to Have Arisen from Random Sampling. *Philos. Mag.* 50, 302 (1900), 157–175.
[23] Kelsey R Fulton, Samantha Katcher, Kevin Song, Marshini Chetty, Michelle L Mazurek, Chloé Messdaghi, and Daniel Votipka. 2023. Vulnerability Discovery for All: Experiences of Marginalization in VulnerabilityDiscovery. *Proc. of the IEEE* (2023).
[24] GhidraJupyter. 2022. Ghidra Jupyter Kotlin. Accessed: 2022.
[25] Comsecuris GmbH. 2018. gbd IDA. Accessed: 2022.
[26] Patrice Godefroid, Michael Y Levin, and David A Molnar. 2008. Automated whitebox fuzz testing.. In *NDSS*, Vol. 8. 151–166.
[27] Munawar Hafiz and Ming Fang. 2016. Game of detections: how are security vulnerabilities discovered in the wild? *Empirical Software Engineering* 21, 5 (2016), 1920–1959. https://doi.org/10.1007/s10664-015-9403-7
[28] hasherezade. 2022. IDA IFL. Accessed: 2022.
[29] Øyvind Hauge, Carl-Fredrik Sørensen, and Reidar Conradi. 2008. Adoption of Open Source in the Software Industry. In *Open Source Development, Communities and Quality*, Barbara Russo, Ernesto Damiani, Scott Hissam, Björn Lundell, and Giancarlo Succi (Eds.). Springer US, Boston, MA, 211–221.
[30] Andrew F Hayes and Klaus Krippendorff. 2007. Answering the call for a standard reliability measure for coding data. *Communication methods and measures* 1, 1 (2007), 77–89. http://dx.doi.org/10.1080/19312450709336664
[31] Jeffrey Heer, Jock Mackinlay, Chris Stolte, and Maneesh Agrawala. 2008. Graphical Histories for Visualization: Supporting Analysis, Communication, and Evaluation. *IEEE Transactions on Visualization and Computer Graphics* 14, 6 (2008), 1189–1196. https://doi.org/10.1109/TVCG.2008.137
[32] Hex-Rays. 2019. Plug-in Contest 2018: Hall of Fame. https://www.hex-rays.com/contests/2018/index.shtml (Accessed 05-30-2019).
[33] Hex-Rays. 2022. IDA. Accessed: 2022.
[34] Shih-Kun Huang, Min-Hsiang Huang, Po-Yen Huang, Chung-Wei Lai, Han-Lin Lu, and Wai-Meng Leong. 2012. CRAX: Software Crash Analysis for Automatic Exploit Generation by Modeling Attacks as Symbolic Continuations. In *2012 IEEE Sixth International Conference on Software Security and Reliability*. 78–87. https://doi.org/10.1109/SERE.2012.20
[35] Alan Jaffe, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, and Bogdan Vasilescu. 2018. Meaningful Variable Names for Decompiled Code: A Machine Translation Approach. In *2018 Conference on Program Comprehension* (Gothenburg, Sweden) *(ICPC '18)*. Association for Computing Machinery, New York, NY, USA, 20–30. https://doi.org/10.1145/3196321.3196330
[36] T.j. Jankun-Kelly, Kwan-liu Ma, and Michael Gertz. 2007. A Model and Framework for Visualization Exploration. *IEEE Transactions on Visualization and Computer Graphics* 13, 2 (2007), 357–369. https://doi.org/10.1109/TVCG.2007.28

[37] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. DIRE: A Neural Approach to Decompiled Identifier Naming. In *2019 IEEE/ACM International Conference on Automated Software Engineering (ASE '19)*. 628–639. https://doi.org/10.1109/ASE.2019.00064
[38] James Andrew Lewis. 2019. The Cybersecurity Workforce Gap. https://www.csis.org/analysis/cybersecurity-workforce-gap (Accessed 08-06-2021).
[39] Robert Lyda and James Hamrock. 2007. Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy* 5, 2 (2007), 40–45.
[40] Alessandro Mantovani, Simone Aonzo, Yanick Fratantonio, and Davide Balzarotti. 2022. RE-Mind: a First Look Inside the Mind of a Reverse Engineer. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/usenixsecurity22/presentation/mantovani
[41] Martyx00. 2021. VulnFanatic. Accessed: 2022.
[42] Brandon (zznop) Miller. 2020. Sega Genesis. Accessed: 2022.
[43] Nalen98. 2021. A List of IDA Plugins. https://github.com/Nalen98/AngryGhidra (Accessed 07-25-2022).
[44] NetDiligence. 2022. *NetDiligence Ransomware 2022 Spotlight Report*. Technical Report. NetDiligence, Gladwyne, Pennsylvania. https://netdiligence.com/wp-content/uploads/2022/05/NetD_2022_Ransomware-Spotlight_1.1.pdf
[45] Timothy Nosco, Jared Ziegler, Zechariah Clark, Davy Marrero, Todd Finkler, Andrew Barbarello, and W. Michael Petullo. 2020. The Industrial Age of Hacking. In *2020 USENIX Security Symposium (USENIX Security '20)*. USENIX Association, 1129–1146. https://www.usenix.org/conference/usenixsecurity20/presentation/nosco
[46] onethat. 2021. A List of IDA Plugins. https://github.com/onethawt/idaplugins-list (Accessed 06-27-2022).
[47] William A. Pike, John Stasko, Remco Chang, and Theresa A. O'Connell. 2009. The Science of Interaction. *Information Visualization* 8, 4 (2009), 263–274. https://doi.org/10.1057/ivs.2009.22 arXiv:https://doi.org/10.1057/ivs.2009.22
[48] Stephan Plöger, Mischa Meier, and Matthew Smith. 2021. A Qualitative Usability Evaluation of the Clang Static Analyzer and libFuzzer with CS Students and CTF Players. In *2021 Symposium on Usable Privacy and Security (SOUPS '21)*. USENIX Association, 553–572.
[49] Hugo (icecr4ck) Porcher. 2020. GameBoy. Accessed: 2022.
[50] Radare. 2019. Radare. https://rada.re/n/radare2.html (Accessed 11-11-2019).
[51] Radare.org. 2022. Radare2. Accessed: 2022.
[52] Hex Rays. 2021. FindCrypt Hex-Rays. Accessed: 2022.
[53] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *2010 IEEE Symposium on Security and Privacy (SP '10)*. 317–331. https://doi.org/10.1109/SP.2010.26
[54] Ben Shneiderman. 2003. The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In *The Craft of Information Visualization*, BENJAMIN B. BEDERSON and BEN SHNEIDERMAN (Eds.). Morgan Kaufmann, San Francisco, 364–371. https://doi.org/10.1016/B978-155860915-0/50046-9
[55] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. 138–157. https://doi.org/10.1109/SP.2016.17
[56] Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna. 2017. Rise of the HaCRS: Augmenting Autonomous Cyber Reasoning Systems with Human Assistance. In *Proc. of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM.
[57] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution.. In *NDSS*, Vol. 16. 1–16.
[58] Anselm Strauss and Juliet Corbin. 1990. *Basics of qualitative research*. Vol. 15. Newbury Park, CA: Sage.
[59] Scott Tenaglia and Perri Adams. 2020. *Edge of the Art in Vulnerability Research*. Technical Report. Two Six Labs.
[60] Vector35. 2019. Vector35/Community-Plugins. https://github.com/Vector35/community-plugins/tree/master/plugins (Accessed 05-30-2019).
[61] Daniel Votipka, Mary Nicole Punzalan, Seth M. Rabin, Yla Tausczik, and Michelle L. Mazurek. 2021. An Investigation of Online Reverse Engineering Community Discussions in the Context of Ghidra. In *2021 IEEE European Symposium on Security and Privacy (EuroS P)*. 1–20. https://doi.org/10.1109/EuroSP51992.2021.00012
[62] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S. Foster, and Michelle L. Mazurek. 2020. An Observational Investigation of Reverse Engineers' Processes. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1875–1892. https://www.usenix.org/conference/usenixsecurity20/presentation/votipka-observational
[63] Daniel Votipka, Rock Stevens, Elissa M Redmiles, Jeremy Hu, and Michelle L Mazurek. 2018. Hackers vs. Testers: A Comparison of Software Vulnerability Discovery Processes. *Proc. of the IEEE* (2018).
[64] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith. 2016. Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study. In *IEEE S&P '16*. 158–177. https://doi.org/10.1109/SP.2016.18
[65] Ji Soo Yi, Youn ah Kang, John Stasko, and J.A. Jacko. 2007. Toward a Deeper Understanding of the Role of Interaction in Information Visualization. *IEEE Transactions on Visualization and Computer Graphics* 13, 6 (2007), 1224–1231. https://doi.org/10.1109/TVCG.2007.70515
[66] Miuyin Yong Wong, Matthew Landen, Manos Antonakakis, Douglas M. Blough, Elissa M. Redmiles, and Mustaque Ahamad. 2021. An Inside Look into the Practice of Malware Analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 3053–3069. https://doi.org/10.1145/3460120.3484759
[67] zaironne. 2015. Snippet Detector. Accessed: 2022.

A TOOL SEARCH PROCESS

Below are the different tool search methods employed to source our data set:

- **Framework-curated lists:** Initially we searched the official framework's Github page and website for any links to user-created plugins or lists of recommended plugins. These lists typically indicate most popular and routinely maintained plugins for the framework.
- **Keyword search:** We next searched Google and Github broadly for *plugins* and *extensions* for each framework. We made use of every combination of the terms *IDA*, *Ghidra*, *Radare*, and *Binary Ninja* with the terms *plugin* and *extension* to find results for each framework. We searched every resulting link for a downloadable plugin and continued on until we repeatedly encountered duplicate plugins. For each keyword search, we reviewed at least the first page of results.
- **Twitter search:** We searched each framework's Twitter page for plugin retweets. We also repeated our keyword searches on Twitter. This method was specifically useful for Ghidra. Ghidra is maintained by the National Security Agency (NSA), so there are no officially approved plugin lists. However, Ghidra's official Twitter page retweets security researchers when they publish a new plugin.
- **Prior tool surveys:** Lastly, we reviewed previously published surveys of reverse engineering tool functionality [5, 53, 55, 59] to find individually published plugins and standalone tools. In particular, Tenaglia et al. [59] provided a thorough overview and offered the most breadth for our search.

B CODEBOOKS

In this section, we give the full definition and counts for the different entries in our codebook. Starting with the input content codes (Table 4), the output content codes (Table 5), the output method codes (Table 6), and then the static (Table 7) and dynamic (Table 8) functionality types.

Code	Definition	Count
Input file		
Binary file	When the tool ONLY takes a binary file as input.	162
Assembly file	When the tool ONLY takes an assembly file as input.	2
Selected area	When the tool operates on a specific area the user can select.	49
Configuration		
Scripts/Strings	Whenever the user defines their own script, code, or values as input.	24
Operation mode	When the user can specify a particular operation mode or set runtime flags for the tool.	42
Dynamic State		
Trace info	When the tool takes trace information as input.	5
Debugger info	When the tool relies on debugging information or functions jointly with a debugger.	1
Memory dump	When the tool take crash dump information or other memory dump information.	1

Table 4: Input Content Definitions: Table of each input type in the codebook listing the entry, its definition as it applies to the tools, and the final count encountered across our data set.

Code	Definition	Count
Hierarchy		
Function signatures	When the tool produces information regarding function names, parameter information, and return information.	151
Variable information	When the tool output produces information regarding variable values or location.	79
New patched code	When the tool modifies the input file to produce new patched code.	23
Selected area	When the tool highlights an area of code / moves the user's cursor to a specific location.	14
Class information	When the tool output produces information regarding class structure.	11
Cross-program		
Cross references	When the tool produces cross reference locations of matched functions / variables / strings.	75
String matching	When the tool produces a list of matched strings based on search parameters.	62
Binary file	When the tool produces a new binary file.	3
Assembly file	When the tool produces a new assembly file.	5
User modifications	When the tool implements user modifications such as renaming all instances of a variable.	7
Tool info.		
Binary data	When the tool output concerns raw binary data or metrics.	3
Debugger info	When the tool produces or modifies debugging information.	15
Decompiler info	When the tool produces or modifies decompiler information.	13
Emulator info	When the tool produces or modifies emulator information.	10
Framework output	When the plugin modifies the framework's output or modifies framework functionality.	17

Table 5: Output Content Definitions: Table of each output content type in the codebook listing the entry, its definition as it applies to the tools, and the final count encountered across our data set.

Codebook Entry	Definition	Count
Text		
Command line	When the tool output is provided on a command line not integrated with any framework.	62
Console log	When the output of the plugin is produced on the integrated console log of the framework.	48
Table	When the output is provided in tabular format.	44
Code View		
Assembly Code Viewer	When the output of the plugin is presented in the assembly code view of the framework.	58
Decompiled Code Viewer	When the output of the plugin is presented in the decompiled code view of the framework.	31
Highlighted segments	When the tool highlights an area of code.	16
File creation / modification	When the tool modifies or creates a new file of any type.	61
Data Vis.		
Graph visualization	When the tool produces a graph or figure of the results.	30
GUI Window	When the output is provided in a pop-up / integrated GUI window of the framework.	28
Image	When the tool produces an image of the data.	1
Launches client plugin	When the plugin launches an external tool to produce output or operate.	13

Table 6: Output Method Definitions: Table of each output method type in the codebook listing the entry, its definition as it applies to the tools, and the final count encountered across our data set.

Codebook Entry	Definition	Count
Heuristic Scan / Pattern matching	When the tool scans the binary file, returning values based on predefined heuristics.	148
Inter Framework Transfer	When the tool enables moving, copying, reporting metadata and connecting frameworks with other frameworks or external tools.	49
Code Visualization	These tools produce visualizations of the binary file, such as control flow graphs.	42
Mod Framework	When the tool changes how REs interact with the framework not specific to any task.	29
Symbolic Execution	These tools emulate dynamic analysis on different paths in a program to identify what parameters on an input will create a specific path to a portion of the code being executed.	18
Disassembler	When the tool extends the existing framework disassembler or functions as a disassembler.	17
Decompiler	When the tool extends the existing framework decompiler or functions as a decompiler.	14
Diff	When the tool identifies differences in two files or selected ranges.	9
Instruction Slicing	When the tool implements machine code slicing to create simpler subsets of instructions.	1

Table 7: Static Functionality Definitions: Table of each static functionality type in the codebook listing the entry, its definition as it applies to the tools, and the final count encountered across our data set.

Codebook Entry	Definition	Count
Fuzzing	These tools provide random program inputs to discover new vulnerabilities or bugs.	28
Debugger	These tools either extend or act as standard debuggers.	25
Hooking / Patching	When the tool facilitates new patches to the binary or allows dynamic hooking.	24
Emulator	When the tool emulates select functions or the entire binary to observe code behavior.	9
Kernel Symbol	When the tool applies kernel symbols to binary or gathers kernel runtime information.	2
Devirtualize Calls	When the tool uses runtime information to devirtualize or deobfuscate function calls.	2

Table 8: Dynamic Functionality Definitions: Table of each dynamic functionality type in the codebook, listing the entry, its definition as it applies to the tools, and the final count encountered across our data set.