I Can SE Clearly Now: Investigating the Effectiveness of GUI-based Symbolic Execution for Software Vulnerability Discovery

YI JOU LI, Arizona State University, USA
ZEMING YU, Arizona State University, USA
JAMES A. MATTEI, Tufts University, USA
ANANTA SONEJI, Arizona State University, USA
ZHIBO SUN, Drexel University, USA
RUOYU WANG, Arizona State University, USA
JARON MINK, Arizona State University, USA
DANIEL VOTIPKA, Tufts University, USA
TIFFANY BAO, Arizona State University, USA

While symbolic execution (SE) can discover software vulnerabilities, it has received limited practical adoption. A key barrier is that SE requires human expertise to understand the program's state and prioritize paths to analyze. Traditionally, users controlled SE through programmatic API calls, but recent tooling now implements graphical user interfaces (GUI). However, it is unclear how these new features affect human-SE performance. To understand this impact, we conducted a controlled experiment where 24 vulnerability discovery experts were tasked with analyzing a binary using an SE tool with either API or GUI-based features. From this study, we identify (1) experts' SE process, and (2) the impact of GUI-based features on human-SE performance. Then we propose recommendations to improve SE tool design.

CCS Concepts: • **Human-centered computing → User studies**; • **Security and privacy → Software and application security**.

Additional Key Words and Phrases: symbolic execution, user study, vulnerability research

ACM Reference Format:
Yi Jou Li, Zeming Yu, James A. Mattei, Ananta Soneji, Zhibo Sun, Ruoyu Wang, Jaron Mink, Daniel Votipka, and Tiffany Bao. 2026. I Can SE Clearly Now: Investigating the Effectiveness of GUI-based Symbolic Execution for Software Vulnerability Discovery. In *Proceedings of Human Factors in Computing Systems (CHI '26).* ACM, New York, NY, USA, 27 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

As society becomes increasingly reliant on software, addressing software vulnerabilities has become a critical focus in cybersecurity. To enhance software protection, researchers have developed several techniques to proactively identify

Authors' Contact Information: Yi Jou Li, yijouli@asu.edu, Arizona State University, Tempe, AZ, USA; Zeming Yu, zemingyu@asu.edu, Arizona State University, Tempe, AZ, USA; James A. Mattei, Tufts University, Medford, MA, USA, james.mattei@tufts.edu; Ananta Soneji, asoneji@asu.edu, Arizona State University, Tempe, AZ, USA; Zhibo Sun, zs384@drexel.edu, Drexel University, Philadelphia, PA, USA; Ruoyu Wang, fishw@asu.edu, Arizona State University, Tempe, AZ, USA; Jaron Mink, jmmink1@asu.edu, Arizona State University, Tempe, AZ, USA; Daniel Votipka, Tufts University, Medford, MA, USA, daniel.votipka@tufts.edu; Tiffany Bao, tbao@asu.edu, Arizona State University, Tempe, AZ, USA.

and mitigate security vulnerabilities. Among these, symbolic execution (SE) is recognized as a particularly effective method [8, 18, 46]. By treating inputs as symbolic values and simulating the program's execution [11], exhaustive SE can comprehensively identify *all possible inputs* that violate a specified security policy, ensuring completeness and soundness [8]. This capability makes SE a popular approach for various security tasks including vulnerability discovery [58, 59, 65, 74], automated testing [15, 16, 55, 75], exploit generation [7, 21], reverse engineering [22, 29], automated patching [64], and root cause analysis [86].

Unfortunately, exhaustive symbolic execution exploration is often considered impractical due to its high computational demands, which results in limited adoption by cybersecurity analysts [83, 84] and developers [82]. A key reason for this computational demand is the *path explosion* problem. To comprehensively evaluate all possible inputs leading to vulnerable states, SE must simulate every potential execution path. As the number of divergent paths grows, SE quickly consumes computational resources at an exponential rate. To address this, researchers have developed methods to operate on *path optimizations*, such as pruning paths or concretizing input values [40, 43]. However, these approaches have proven challenging to generalize and the issue remains an open research problem [9].

In practice, overcoming this path explosion problem requires a human-in-the-loop to optimize SE by prioritizing and eliminating paths they believe will direct execution toward a potential vulnerability. This human-driven path prioritization and pruning has traditionally been performed using application programming interface (API) calls made in the interactive command line interface or through the development of analysis scripts; these calls take effect during symbolic program's execution. However, graphical user interfaces (GUI) for symbolic execution has begun to be implemented in tools, potentially allowing SE to be more usable [10, 33, 41, 42, 44, 69, 79]. Rather than requiring API calls to be written in the source code and re-compiled, users interact with SE via a GUI alone [10]. However, it remains unclear whether new user interfaces alone can improve experts' handling of path explosion and enhance the overall performance.

Prior work has shown mixed results in other domains when comparing expert use of GUIs to APIs and command line interfaces (CLI). While GUIs may be easier for novices, the command line can be more expressive and flexible for those who have mastered the syntax [68, pg.317-341]; for example, prior work in software engineering [45, 49] and cybersecurity tasks [81] showed experts perform more effectively using a CLI. Without a clear understanding of *whether and how* GUI-based features impact path explosion in SE, future tools risk either implementing these interfaces ineffectively, introducing interfaces that get in the experts' way, or overlook GUIs' benefits.

In this paper, we take the first step toward understanding how GUI-based SE features can impact vulnerability discovery. Specifically, we ask:

RQ1 What is the high-level workflow SE experts conduct during vulnerability discovery? (Section 5)

RQ2 How does SE workflow and performance differ depending on whether experts' tools are equipt with API-based or GUI-based features? (Section 6)

To answer these questions, we conducted a user study with 24 vulnerability discovery experts, well-versed with API-based symbolic execution. Participants were asked to perform a vulnerability discovery task of realistic length and complexity and were randomly assigned to either use an API-based SE tool or a GUI-based SE tool which is based on the same SE engine. We then evaluate how participant's behaviors, effectiveness in task completion, and efficiency of path prioritization operations varied due to the use of the GUI-based SE features. We performed a mix of qualitative and quantitative analyses to understand how each feature in the GUI-based tool impacted user behavior.

Based on these results we find that (1) The GUI's path optimization operation on target program feature and SE progress pause and resume feature changes experts' workflow in setting up the first SE pass and adjusting path exploration priority, (2) Multiple GUI features (e.g., initializing symbolic execution states via pop-up dialogues and configurable buttons) reduce the cognitive and operational overhead required to perform equivalent tasks through APIs. (3) participants who used GUI-based features perform path prioritization operations *more frequently*, and (4) this increase in activities led to more effective path exploration and vulnerability discovery.

We also offer several actionable recommendations for SE tool designers. These recommendations include strategies to enhance the effectiveness and correctness of path optimization, improve user interactions during vulnerability triage, and introduce interpretive tools for better exploration of symbolic states.

In summary, our work makes the following contributions:

- We introduce and summarize the features of existing GUI-based symbolic execution tools.
- We conduct a human-subject experiment to assess and study the impact of GUI features on vulnerability discovery experts' behaviors and performance.
- Our experiment shows that different GUI-based symbolic execution features do impact users' workflows and can lead to better performance, with higher correctness in less time.
- We propose several suggestions for the future development of, and research on, GUI-based symbolic execution.

To ensure transparency and reproducibility, we provide supplementary materials (task descriptions, participant surveys, and experiment code) in a public repository [6].

2 Background and Related Work

2.1 Symbolic Execution

Symbolic Execution is a program analysis technique that executes a program in an emulated environment where part of the program context such as program variables and file descriptors are considered symbolic [46]. Through this execution, the state of registers and memory, and the constraints on those variables are tracked. Importantly, whenever a conditional branch is reached, execution forks and follows *both paths*, saving the relevant potential branch conditions as a constraint, and continues execution [70].

KLEE [15], one of the most influential symbolic execution tools, leverages symbolic execution to systematically explore execution paths, uncovering critical memory corruptions such as buffer overflow, use-after-free, and integer overflow vulnerabilities in complex software. Its successors extend the model to support low-level memory reasoning (KLEE-ram [78]), past-sensitive pointer analysis (KLEE-pspa [77]), and symbolic object sizing (KLEE-symsize [76]).

S2E [23] extends symbolic execution to complex, full-stack systems by selectively symbolically executing individual program components. SYMBION [35] interleaves symbolic and concrete execution to improve path recovery. Performance-oriented systems such as QSYM [90], SymCC [62], and SymQEMU [63] improve scalability through binary-level instrumentation and hybrid execution. angr [70] provides a modular symbolic execution framework integrated with binary analysis capabilities, supporting vulnerability discovery and exploit generation.

Symbolic execution has been widely adopted for vulnerability discovery in various domains. For instance, it has been used in traditional software applications [17], web applications [48, 67], mobile applications [34, 55], IoT devices [19, 30], cyber-physical systems [31], and smart contracts [39, 57, 73]. These efforts demonstrate the flexibility and power of symbolic execution in uncovering memory corruption, logic errors, and input validation issues.

Table 1. **Representative GUI-based SE Tools and Features** — We present the a set of representative GUI-based symbolic execution tools and the features included or excluded from each.

GUI-Based SE Tool	Initial State Config.		Process Control				Process Visualization		Avoid/Targeted Instruction Specification		Symbolic State Investigation		
	Start Address	Symbol Source	Pause	Resume	Stop	Step	Program Coverage	Exploration Tree	Program View	Stash View	Mark Symbolic Operands	Concretize Input	Constraint
Ponce	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✓	✗	✗
ManticoreUI	✓	✓	✓	✓	✓	✓	✗	✗	✓	✗	✗	✓	✗
SENinja	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓
angr-management	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓
SED	✗	✓	✓	✓	✓	✓	✗	✓	✓	✗	✗	✓	✓
GAIT	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✓	✓
SEViz	✗	✗	✓	✓	✓	✓	✗	✓	✓	✗	✗	✗	✗

Nevertheless, ongoing research efforts still focus on improving the practicality of symbolic execution-based vulnerability analysis in real-world software programs. In this paper, we focus on whether GUI-based symbolic execution can mitigate the issue and impact the performance of symbolic execution in a real-world scale vulnerability discovery task.

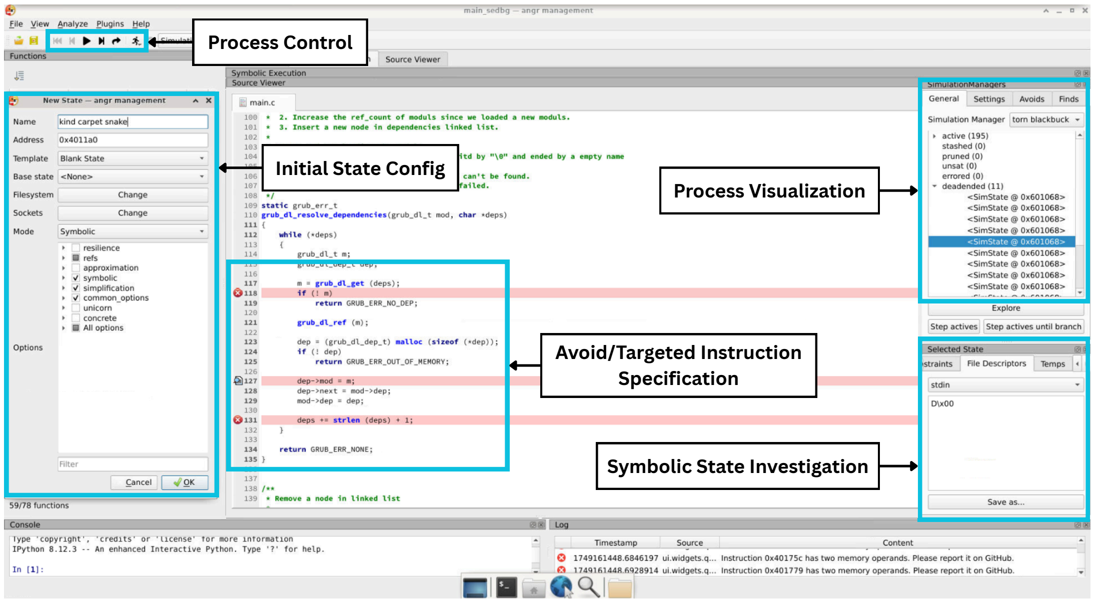
2.2 GUI-based Symbolic Execution

GUI-based symbolic execution has gained significant popularity within the community over the past decade [1, 2, 10]. It offers a potentially more intuitive interface over API-based tools, allowing users to easily interact with symbolic execution engines, visualize execution paths, manage symbolic states, and interpret analysis results without needing extensive knowledge of underlying engine-specific details and API semantics.

Table 1 presents representative GUI-based symbolic execution tools along with their specific GUI design features. The table categorizes these features according to their purpose in symbolic execution operations and summarizes how these features are implemented across seven representative GUI-based symbolic execution tools: Ponce [44], ManticoreUI [79], SENinja [10], angr-management [69], SED [41], GAIT [33], and SEViz [42].

Specifically, we classify the GUI-based symbolic execution design features as follows

- **Initial Symbolic State Configuration.** This feature class supports users to customize the initial state for symbolic execution, such as selecting the starting location and specifying symbol sources.
- **Symbolic Execution Progress Control.** Similar to debuggers' execution control, this class includes pausing, resuming, stopping, and stepping the current symbolic execution process.
- **Symbolic Execution Progress Display.** This class of features visualizes the current progress of symbolic execution while exploring the program. Existing tools have two ways to visualize the exploration progress: tree view and stash view. For tree view, tools such as SED [38, 41] and SEViz [42] display all explored symbolic program states as a tree where the root is the initial state and the children nodes are the split states of the parent node due to conditional jump. For stash view, it only shows the states considered "special" (e.g., associated with a target program location) or currently being explored. Existing GUIs with this feature did not support a direct access of explored states.


main_sedbg — angr management
File View Analyze Plugins Help
Functions
Process Control
Source Viewer
Symbolic Execution
Initial State Config
Process Visualization
Avoid/Targeted Instruction Specification
Symbolic State Investigation
SimulationManagers
General Settings Avoids Finds
Explore
Step actives
Step actives until branch
Selected State
straints File Descriptors Temps
Save as...
Cancel
OK
59/78 functions
Console
Log

New State — angr management

Name	kind carpet snake
Address	0x4011a0
Template	Blank State
Base state	<None>
Filesystem	Change
Sockets	Change
Mode	Symbolic
Options	resilience, refs, approximation, [✓ symbolic], [✓ simplification], [✓ common_options], unicorn, concrete, All options

main.c

```
 *  2. Increase the ref_count of moduls since we loaded a new moduls.
 *  3. Insert a new node in dependencies linked list.
 *
                                    itd by "\0" and ended by a empty name

                                     can't be found.
                                    failed.
 */
static grub_err_t
grub_dl_resolve_dependencies(grub_dl_t mod, char *deps)
{
    while (*deps)
    {
        grub_dl_t m;

        m = grub_dl_get (deps);
        if (! m)
            return GRUB_ERR_NO_DEP;

        grub_dl_ref (m);

        dep = (grub_dl_dep_t) malloc (sizeof (*dep));
        if (! dep)
            return GRUB_ERR_OUT_OF_MEMORY;

        dep->mod = m;
        dep->next = mod->dep;
        mod->dep = dep;

        deps += strlen (deps) + 1;
    }

    return GRUB_ERR_NONE;
}


/**
 * Remove a node in linked list
```

Simulation Manager: torn blackbuck

```
active (195)
stashed (0)
pruned (0)
unsat (0)
errored (0)
deadended (11)
        <SimState @ 0x601068>
        <SimState @ 0x601068>
        <SimState @ 0x601068>
        <SimState @ 0x601068>
        <SimState @ 0x601068>
        <SimState @ 0x601068>
        <SimState @ 0x601068>
        <SimState @ 0x601068>
        <SimState @ 0x601068>
```

stdin

```
D\x00
```

```
Type 'copyright', 'credits' or 'license' for more information
IPython 8.12.3 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

```
Timestamp           Source           Content
1749161448.6846197  ui.widgets.q...  Instruction 0x40175c has two memory operands. Please report it on GitHub.
1749161448.6928914  ui.widgets.q...  Instruction 0x401779 has two memory operands. Please report it on GitHub.
```

Fig. 1. **angr-management SE GUI** — Here we show the graphical features contained within angr-management.

- **Avoid/Targeted Instruction Specification.** This class includes all actions an expert can take through the interface to prioritize a particular path to mitigate the path explosion issue occurring during the symbolic execution's path exploration process. Path explosion is an inherent challenge with symbolic execution as each program branch introduces a new path and the system must maintain state for each possible path in memory. Path prioritization allows users to only maintain state for certain paths, reducing memory needed, however, with a loss of analysis precision. Different tools provide different ways for users to prioritize paths. SENinja supports program location-based prioritization, which allows users to specify targeted program locations or locations to be avoided in path exploration. SED supports state-based prioritization, which allows users to pause the exploration process, select one symbolic state and continue only the exploration of the selected state. GAIT and angr-management support both program location-based and symbolic state-based prioritization, and they support the selection of multiple symbolic states and continued exploration.
- **Symbolic State Investigation.** When the user is interested in a particular symbolic state, GUI-based SE allows users to access the details associated with the state, including the associated path constraints and the symbolic and concrete program context. Some tools such as ManticoreUI concretize the symbols and provide a viable solution that satisfies all constraints, i.e., a concrete input which causes the program to execute to the selected state.

In this study, we use the SE engine angr, and its supported GUI, angr-management, to investigate the differences between API and GUI workflows and performance in SE tools. While several SE frameworks for binary analysis exist, we select angr as it has wide-community use and supports step-by-step interactions such as inspecting the symbolic execution process and performing path prioritization interactively. As the GUI counterpart of angr, we chose angr-management as the representative GUI-based SE tool for a fair comparison. It also offers the most features among the GUI-based vulnerability discovery-oriented SE tools we reviewed [10, 38, 41, 42]. The choice of angr and angr-management not only provide external validity, but also comprehensively evaluate our features of interest.

As shown in Figure 1, angr-management design the interfaces for the five above-mentioned features in the following way.

- **Initial Symbolic State Configuration.** To run symbolic execution in angr, users need to draft a script that contains API function calls to create SE states and other required functions. In this GUI-based tool, these API actions are replaced with a state creation list, which allows users to customize the state name and starting address, modify state templates and sockets, and select templates and options as needed.
- **Symbolic Execution Progress Control.** This feature is designed as a control panel, which allows users to start, stop, pause, continue, and step in the symbolic execution.
- **Symbolic Execution Progress Display.** On the right side of the interface, the GUI lists the simulation managers, enabling users to track the current states in different stashes throughout the simulation.
- **Avoid/Targeted Instruction Specification.** This design allows users to prioritize or withdraw a specific location by marking the line in either the source code or the disassembly code.
- **Symbolic State Investigation.** Users can obtain details associated with symbolic states, such as register lists, file descriptors, and memory locations, by clicking the different tabs in the selected state window.

2.3 Human Factors in Vulnerability Discovery

Finally, while to our knowledge there is no prior work focusing on the human factors of symbolic execution, several recent studies have explored the influence of human factors in various fields of vulnerability discovery. Ploger et al. investigated the challenges beginners faced when setting up and running a fuzzer and static analyzer [60], potentially supporting the idea that tools without user-friendly interfaces affect subsequent use. Yakdan et al. showed DREAM++, a usability optimized decompiler which relied on heuristic-based transformations to enhance readability, improved task performance compared to it's less usable counterpart [87]. In the subfield of reverse-engineering, Ceccato et al. [20] conducted a study involving three penetration testing teams to understand protected programs and discover and exploit identified vulnerabilities. Bryan et al. examined the human sense-making process of reverse engineering [13]. Votipka et al. iterated on this by developing a detailed model describing specific approaches used throughout the process [83]. Similarly, Mantovani et al. observed the different strategies of novices and experts while completing reverse engineering tasks to describe their mental models [51]. Mattei et al. [52] builds on these work, noting that tools do not always fit existing user processes. Compared to these works that retroactively evaluate the effectiveness of a tool and whether it clashes with users processes, we focuses on understanding the RE's current processes to develop guidelines for how tools can support human cognition.

Most similarly, Shoshitaishvili et al. [71] introduced a novel human-machine interface to enhance the performance of state-of-the-art fuzzing techniques. The interface records users' interaction with the testing program and uses it to diversify the seeds for fuzzing and thus enhancing fuzzing performance. This suggests supporting human-interaction in vulnerability discovery can improve vulnerability discovery performance in real-world applications. Compared to this work, we focus on whether a graphical user interface improves operator effectiveness in SE – a process in which expert users must proactively decide when and how to choose execution paths, and proactively contend with memory and computation limits. Compared to fuzzing methods in Shoshitaishvili et al. [71], the users' operations in an SE

```
int dl_unref (dl_t mod) {
  dl_dep_t dep;
  // [Vulnerability: Pointer Use]
  // The vulnerability is triggered here, where module A (mod->dep, when mod is C) is used after it is
      freed by mistake
  for (dep = mod->dep; dep; dep = dep->next)
    dl_unref (dep->mod);
  // [Vulnerability: Root Cause] mod->ref_count only decreases by 1 for multiple dependents, and it
     still decreases when the module has no dependents
  return --mod->ref_count;
}
int mini_cmd_rmmod (char *name) {
  dl_t mod;
  mod = dl_get (name);
  if (! mod)
    return ERR_BAD_ARGUMENT;
  if (dl_unref (mod) <= 0)
    // [Vulnerability: Pointer Free]
    // When module A's reference count (mod->ref_count) is mistakenly decreased to 0, it will be
     mistakenly freed
    dl_unload (mod);
  else
    printf("Can't rm %s: some mods depend on it", name);
  return 0;
}
int main() {
... // buff is gathered from unsafe user input
grub_mini_cmd_rmmod(buff);
...
}
```

Listing 1. **Vulnerable Code Analyzed In Study** — We present code snippet shows the vulnerability used in the study. The full program is provided in the supplementary materials [6]

tools are arguably more important to the ultimate success (as SE is much less scalable than fuzzing), and thus more essential to support. Also, while prior work that focuses on non-expert users, our study examines expert practitioners. Given the steep learning curve of symbolic execution [80], only experienced users can fully leverage its capabilities for vulnerability discovery. Focusing on experts is thus essential to accurately assess the benefits that GUI support can bring to symbolic execution workflows.

3 Methods

To investigate the workflows SE experts conduct during vulnerability discovery (**RQ1**), and how their workflows and performance differ depending on whether they use an API- or GUI-based features (**RQ2**) we run a controlled user-study.

3.1 Study Design

Participants were randomly assigned to either use an API- or a GUI-based SE tool (angr and angr-management, respectively), and asked to discover inputs that would crash a vulnerable piece of software. To ensure meaningful results, we (1) carefully designed the vulnerability discovery tasks to be both practical in a user study, yet of realistic length and complexity, and (2) chose a SE framework and GUI that was both common in practical use, and allowed us to evaluate our required suite of features.

Vulnerability Discovery Task. We designed the vulnerability discovery task by considering two main factors. *First*, the task length was chosen to balance realism with the practical constraints of participant time. Vulnerability discovery on real software can take weeks to complete [84, 87], which makes it infeasible to recruit experts for such long periods. However, prior work observed professionals break their program analysis process during vulnerability discovery into three phases: 1) an overview to identify code segments (e.g., a single function or set of blocks of code) of interest, 2) quick scans of those code segments, and 3) focused experimentation to answer specific questions about those code segments [83]. This prior work also found professionals reported using SE tools only in the latter two phases on smaller code segments. We therefore designed our task to mimic the size of such a review, which we believe enables insights into real-world discovery processes while remaining feasible for highly specialized participants. The time constraint also impacts the complexity of the task code. A primary cause of path explosion in symbolic execution is *loops and recursion*. Excessive iteration would trigger catastrophic path explosion and force participants to spend prohibitively long on the task even with optimizations. To ensure exploration could complete within a reasonable time, we designed the task to contain a manageable number of loops and recursions. We pre-tested the task with both unoptimized and optimized SE runs. Without optimization, the analysis suffered from path explosion and exceeded the time limit, whereas with optimization, the run completed in five minutes. This design ensured that tasks did not become trivial but still required meaningful optimization strategies to finish within the study timeframe.

Second, we focused on memory corruption vulnerabilities, a class that is both prevalent in practice and major to SE research [18]. We specifically investigated heap corruption, one of the most common subcategories of real-world memory corruption vulnerabilities [28]. One possible design would be to study all memory corruption subcategories. However, since popular SE engines treat memory generically and most path prioritization techniques are memory-agnostic [10, 15, 23, 85], we considered heap corruption to be representative. We leave type-specific SE studies to future work.

Following these considerations, we designed our experimental task based on CVE-2020-25632 [27], a heap corruption vulnerability in GNU GRUB2, which is widely used as the default boot loader and manager in Ubuntu systems[25]. We trimmed the source code and turned it into a program with 279 lines of code. The core of the task is shown in Listing 1. This program mimics kernel module management in operating systems. In this program, the user has indirect control over `buff` and the amount of times `mini_cmd_rmmod` is called (line 25). The root cause of the vulnerability occurs in function `dl_unref`. The module's dependents counter `ref_count` does not decrease properly with the number of dependent being unlinked from the module. As a result, pointers already freed can still be considered exists and then used in the rest of the execution. To trigger this vulnerability, an SE engine needs to execute the program to invoke function `mini_cmd_rmmod` for multiple times with the correct module name. As the code contains an infinite while loop, the path explosion problem will be triggered if the participant does not successfully optimize the SE engine; there the participant must identify the functions to correctly avoid, and to correctly explore to trigger the vulnerability.

SE Tooling & Environment Setup. To evaluate graphical interfaces' effect on SE effectiveness, we assigned participants to an SE interface that was either GUI-based or API-based. Specifically, we used angr as our API-based tool and angr-management (a GUI built on the angr SE engine) as our GUI-based tool. We opted for angr as the underlying SE engine [85] for several reasons. *First*, to ensure external validity, we used an existing system, well-known to our participant pool with publicly available documentation rather than creating our own. This also helped avoid learning effects [50, pg. 177-180] an unfamiliarity interface would introduce to our results. *Second*, unlike other SE engines, angr has an existing GUI alternative, angr-management, which incorporates all the features of angr, allowing for an

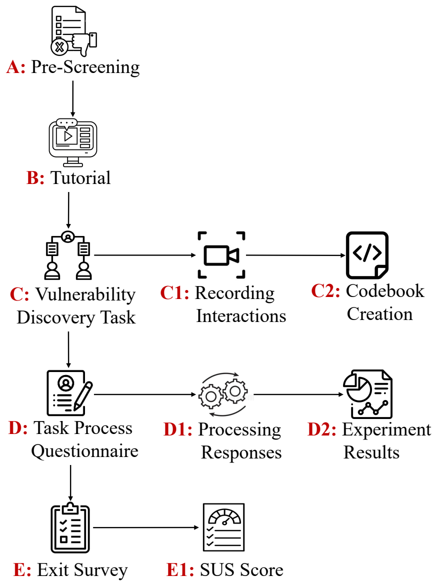


Fig. 2. **Study Procedure** — We present the procedure each participant followed, and the resulting analysis performed on data gathered in each step.

apples-to-apples comparison. Additionally, angr-management incorporates all the graphical features that have been implemented or proposed in other SE tools (see Section 2.2 and Table 1); thus, this serves as a useful real-world probe to understand the impact of these features.

We set up a controlled environment that contains the vulnerability discovery task program and the participant's assigned SE interface. To capture both the SE users' interactions and the SE engine's internal behavior during the experiment, we recorded the entire session using a screen recorder and an angr logging plugin, both running unobtrusively in the background of the controlled environment.

3.2 Recruitment

We broadly advertised our experiment across the Pwn2Own and Capture the Flag competition (CTF) communities. Pwn2Own is a reputable hacking competition where security experts exploit software vulnerabilities for rewards [3]. Capture the Flag (CTF) competitions challenge teams to solve cybersecurity puzzles, by exploiting vulnerable systems that simulate real-world attacks and defenses, fostering skills in ethical hacking [24, 53]. For each of these communities, there are several relevant Discord and Slack channels and we posted recruitment messages across these channels. We also recruited participants through public posts on X (formerly Twitter). Cybersecurity researchers commonly use X to share real-time threat intelligence, disclose vulnerabilities, and stay updated on emerging security trends [56]. In the recruitment advertisement, we explained our research topic, the experiment procedure, and the compensation, and included a link to the pre-survey. The full recruitment message can be found in our supplementary materials [6]. All participants who completed the study received a $50 Amazon giftcard.

3.3 Study Procedure

The study utilized a five-step procedure, ranging from pre-screening to the final usability evaluation. All participants followed this procedure, as shown in Figure 2:

Pre-screening (Step A): Participants were initially screened to ensure they met the study's criteria, which were: a) having more than three years of experience in vulnerability discovery, b) being employed in cybersecurity-related roles or academia, and c) correctly identifying the vulnerability in the skill assessment task. To do this, we administered a pre-screening survey (included in the supplementary materials [6]) consisting of background questions and a vulnerability discovery skill assessment task. The skill assessment task was a 20-line code snippet that contained the same type of vulnerability as the experimental task described in Section 3.1. If participants could identify the line of the code that contained the vulnerability, and/or describe how they could patch the vulnerability, they were eligible to complete the main study. Furthermore, to prevent learning effects from biasing our results, all participants needed to be familiar with our base tool, angr. Thus we further restrict participant to those who have have self-reported experience using angr's symbolic execution engine.

Tutorial (Step B): To ensure participants had an equivalent understanding of how to interact with their assigned tool, all participants were provided either an API, or GUI-tutorial for their corresponding interface. As shown in Section 2.2 and supplementary materials [6], we mimicked the tutorial provided in angr's online documentation[5], and walked each participant through equivalent introductions describing how to use their assigned interface. Additionally, API-assigned participants were also provided an API cheatsheet to easily remember basic commands. To ensure understanding, we also provided participants three practice tasks to try the tools on. Participants could take as much time as needed to review these materials and started the task when they were ready.

Vulnerability Discovery Task (Step C): After the tutorial, participants were randomly assigned one of two tool conditions: 1) GUI-based symbolic execution (angr-management) or 2) API-based symbolic execution (angr). They were then presented with the vulnerability discovery task on a remote desktop, which was recorded throughout the experiment. During this phase, we recorded their interactions *(Step C1)*, capturing how they used the tool to find vulnerabilities.

Task Process Questionnaire (Step D): Upon completing the task, participants filled out a questionnaire detailing how they use the designated SE tool to discovery the vulnerability. The full questionnaire can be found in the supplementary materials [6].

Exit Survey (Step E): Finally, participants completed an exit survey (included in the supplementary materials [6]) that asked about their perception of their assigned tool's usability using the System Usability Scale (SUS) *(Step E1)* [12]. Participants also provided feedback on features they liked and disliked, as well as areas for tool improvement.

3.4 Pilot Study

Prior to the full study, we fine-tuned the primary vulnerability discovery task via two pilot studies with 16 and 17 participants. After the first pilot, we made two significant changes. First, participants were originally given CVE-2019-25013, a relatively simple vulnerability contained in 99 lines of code; however, some participants were able to finish the task without having encountered any path explosion. As path explosion is both a common real-world issue faced by SE analysis, and we wished to evaluate the impact interface makes in such scenarios, we adjusted the task to a more complicated vulnerability, CVE-2021-20232 with 136 lines of code. Second, several participants were confused by the initial GUI tutorial which focused on introducing each button and option in the menu; we adjusted this to map the tutorial more directly to the angr API the expert participants were already familiar with. After implementing these changes, we ran a second pilot study; however, we still observed participants solving the challenge without needing to mitigate path explosion and again adjusted the task to the current version described in Section 3.1. We then recruited 8 additional participants to test this revision, and after finding that all participants understand had a clear understanding of the GUI, and the CVE required all participants to encounter and mitigate path explosion, we affirmed our study design, included the 8 participants in our final sample, and continued recruiting our full sample without alteration.

3.5 Analysis Methods

To understand how GUI-based SE's design can impact experts' vulnerability discovery practices and performance, we adopt a mix of qualitative and quantitative methods, comparing participants assigned to the GUI- or API-based group. To identify themes in the ways the experts use SE tools and how their SE tool's interface — GUI-based or API-based — impacted participant practices when attempting to identify a vulnerability (*Step 2*), we performed an iterative open coding [26]. Two researchers collaboratively manually reviewed 10 participants' tool interactions from video recorded during their session to develop an initial codebook, which is included in the supplementary materials [6]. During this stage, we followed an inductive approach, allowing themes to arise from the data. Using the initial codebook, the two researchers then independently coded participant data in 3 participant batches. After each round, the researchers compared codes, discussed disagreements and reached a consensus. We also made updates as necessary to codebook to refine the codes. We chose not to measure inter-rater reliability, as suggested by best practice [54], as we only report the set of themes identified and do not quantify these results. Finally, we performed axial coding to identify relationships within and between codes and identify higher level themes [26, pg.123-142] that which we report here.

3.6 Ethics

This study was approved by the institutional review board of the institution leading this project. Furthermore, we ensure the privacy and autonomy of our participants through several mechanisms. We anonymize all participants identities and assign them random IDs. Our study does not collect any video or audio of the participants themselves. Participants log in to our remote desktop server to perform the tasks, and the remote desktop server records only the remote desktop and mouse and keyboard input, which is isolated from the participants' local machines. The remote desktop server does not record any other information about the participants. Prior to beginning the study, all participants provided informed consent after having the study and data collection described to them in detail.

3.7 Limitations

In this study, we make several design choices necessary to conduct our study, but that impact our results' interpretation. *First*, we focused on a single memory vulnerability task to ensure a consistent baseline across our small expert participant pool (Section 3.1). While this made it easier to isolate the impact of different interfaces, and provides an important first step for assessing tool performance, ultimately understanding how interfaces generalize across different tasks is left for future work. *Second*, real-world vulnerability discovery can take days or weeks. This timeline is not practical in a research setting as it likely will produce bias from participant dropouts. Also, this scale of real-world problems would make our measurement of path effectiveness impossible as it relies on the ability to perform a SE execution without constraints as described in Section 3.1. In fact, in real-world software reverse engineering tasks, prior work has shown professionals typically do not use SE to analyze the full program, but instead typically usually

Table 2. **Participant Demographics** — We present our participants' reported demographics by their assigned tool condition.

Group	API participants	GUI participants	**Total**
Gender			
Male	9	5	**14**
Female	0	0	**0**
Non-binary / Third gender	0	0	**0**
Did not disclose	4	6	**10**
Age (Year)			
18-24	6	0	**6**
25-34	3	4	**7**
35+	0	0	**0**
Did not disclose	4	7	**11**
Prior Vulnerability Discovery Experience			
0	0	0	**0**
1-5	5	4	**9**
6-10	6	6	**12**
11-15	0	1	**1**
16-20	1	0	**1**
21-25	1	0	**1**
26+	0	0	**0**
Location			
North America	3	6	**9**
Europe	5	1	**6**
Eastern Asia	1	0	**1**
South-Eastern Asia	2	0	**2**
Did not disclose	2	4	**6**
Total	13	11	**24**

use tools like SE on small parts of the code [83]. Therefore, we employed a modified task to mimic this real-world use and reduce the program complexity. While this does not capture long-term workflows, we are still able to capture challenges in configuration and basic use, again offering an important first step toward improving SE, revealing design themes that should be assessed in future work. *Third*, since our study examines only one SE tool and one GUI implementation, we can only speak to their specific impact on success and efficiency; still, we argue that finding any effect from a GUI system highlights the potential of such systems and is a meaningful contribution (Section 6.1). *Forth*, in order to balance the difficulty and the solving effort needed for participants, we simplified our task logic and program length. Based on this necessary choice, our findings may have limited generalizability. However, we want to clarify that this simplification does not impact the core challenge of the task. Simplifying the program does not necessarily mean we reduced the difficulty of vulnerability finding. We still preserved the built-in challenge of discovering the vulnerability the potential path explosion problem in the program. *Fifth*, to avoid interrupting participants, we focused on observing their natural behavior. Future work could investigate their reasoning and conduct a think-aloud study.

4 Participants

Using our three recruitment channels, 520 individuals completed the pre-survey. 29 respondents meet our eligibility criteria and were invited to participate in the main study. Finally, we removed 5 participants from our final data set who did not use their assigned tool.

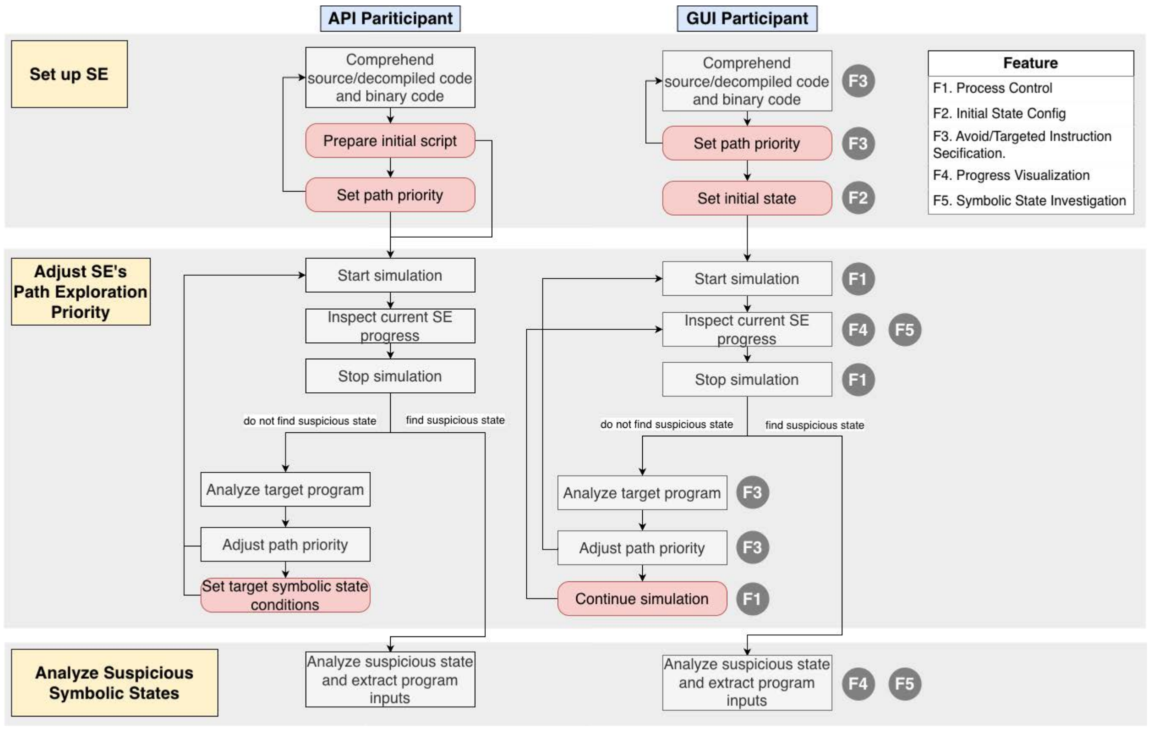

API Pariticipant
GUI Participant
Set up SE
Comprehend source/decompiled code and binary code
Prepare initial script
Set path priority
Set initial state
Feature
F1. Process Control
F2. Initial State Config
F3. Avoid/Targeted Instruction Secification.
F4. Progress Visualization
F5. Symbolic State Investigation
Adjust SE's Path Exploration Priority
Start simulation
Inspect current SE progress
Stop simulation
do not find suspicious state
find suspicious state
Analyze target program
Adjust path priority
Set target symbolic state conditions
Continue simulation
Analyze Suspicious Symbolic States
Analyze suspicious state and extract program inputs
F1
F2
F3
F4
F5

Fig. 3. **Participants' SE Workflow** — Each workflow phase is shown with a yellow box indicating the label and with a gray box encompassing all the steps associated with that phase. Steps are shown in light gray sharp-cornered boxes or red rounded-corner boxes and divided into the API participants' (left) and GUI participants' (right) workflows. The red rounded-corner boxes indicate differences between the two tool groups' workflows which will be discussed in Section 6. Gray circles indicate the steps each GUI feature is used in.

In total, we obtained valid data from 24 participants. Table 2 shows the breakdown of participants' reported gender, age, prior experience with vulnerability discovery, and location. Of the participant who disclosed their demographics, all were male (N=14), younger (18-34, N=13), and most were from North America (N=9) or Europe (N=6). Over half (N=14) of our participants had at least 6-10 years of vulnerability discovery experience. All participants reported having some prior experience using angr. Our sample is clearly demographically skewed, but we expect this because the vulnerability discovery population is similarly skewed with prior large-scale samples showing the community of vulnerability discovery experts is predominantly male, young, and in the earlier stages of their careers [4, 14, 36, 37]. This is problematic for tool design as we seek to grow the vulnerability discovery community and future work tool development research should take steps to recruit from marginalized populations, following the example of Fulton et al. [32]. However, our work presents an important first step in investigating the problem. Similarly, our sample differs geographically from the broader vulnerability discovery community, which has a larger population in South-East Asia. While we expect many of our results will likely generalize as the tools would remain the same across locations, there may be cultural or experiential differences across these communities that should be studied in future work.

5 SE Workflow Analysis (RQ1)

In this section, we provide a high-level workflow analysis of how our participants conducted SE. As mentioned in Section 3, we divide our participants into two groups based on which tool interface they were assigned. To make their tool context clear, from this section onward, we refer to participants assigned to the API- or GUI-based tool as "API participants", and "GUI participants", respectively. Participant IDs will start with A for API participants and with G for GUI participants.

We observed that there were consistent steps in the workflow across both tool groups. In particular, we found all participants followed the three main phases: (1) *setting up SE*, (2) *adjusting path exploration priority*, and (3) *analyzing suspicious symbolic states*. We also observed participants in both tool groups completed a similar set of steps in each phase, though the order, number of iterations through, and exact execution varied due to the differences in features between tools. We describe these common steps in this section, then discuss the variation between tool groups in Section 6. Every participant completed each listed phase and, unless otherwise specified (i.e., with a "n=X" given), every participant completed each step discussed. Each phase and step are summarized in Figure 3.

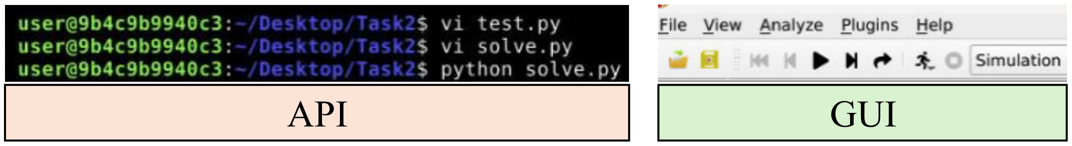
5.1 Setting up SE

Participants' first step in using SE to identify vulnerabilities was to set up symbolic execution by initializing a symbolic program state and configuring exploration options. In this phase, they first needed to comprehend the program by reading its functions, structure, and logic. Next, they determined the symbolic inputs or portions of memory to include in the initial symbolic state, establishing the starting point for the symbolic engine. Some of them (n=7) specified exploration options as well, such as whether to execute library code symbolically or enable the Unicorn engine which concretely executes states when no symbolic information is included, improving simulation performance as state information does not need to be tracked during concrete execution and this limits unnecessary path exploration. During this phase, a subset of participants (n=12) further performed path prioritization, selecting addresses in the source or disassembly code to be prioritized or avoided in the simulation, allowing some initial optimization to focus the tool based on their expectations. After completing this setup step, participants were ready to begin executing the SE process.

5.2 Adjusting path exploration priority

After the symbolic execution pass began, participants monitored its progress to decide whether they needed to interrupt the process and adjust path prioritization–that is, to add new program locations and path prioritizations for the exploration to avoid or target. This prioritization is necessary as it was computationally infeasible for the SE process to assess all possible paths due to path explosion at each branch. That is, the program must maintain an additional copy of program state at each branch or loop in the program to represent the constraints necessary for execution to follow the path to either side of the branch. This prioritization introduces a challenging balance for the user between process efficiency and accuracy as they try to direct the SE process down the most likely paths.

After executing the SE process, if no suspicious states were identified, this could mean their path prioritization caused the SE process to prune paths that led to suspicious states. Thus, participants would next return to the source code to analyze the paths selected in the previous run and then decided which paths should be added or removed for the next simulation. This adjustment process was iterative and continued until one or more suspicious symbolic states were identified during exploration.



```
user@9b4c9b9940c3:~/Desktop/Task2$ vi test.py
user@9b4c9b9940c3:~/Desktop/Task2$ vi solve.py
user@9b4c9b9940c3:~/Desktop/Task2$ python solve.py
```

Fig. 4. **Process Control Interfaces** — We present the tool API- and GUI interfaces for the Process Control feature (F1).

5.3 Analyzing suspicious symbolic states

After identifying one or more suspicious states, i.e., states that violate security policies and could potentially trigger a vulnerability, participants analyzed the constraints on the program inputs that could lead to the vulnerable states. Specifically, participants needed to determine whether these constraints were satisfiable, i.e., that there exists an input which could meet the constraints. For example, if the constraints on input reaching a specific state were [$x > 5$ AND $x < 3$] for the input value x, this is not satisfiable. However, a constraint [$x > 5$ AND $x > 3$] is satisfiable by any number greater than 5.

To assess constraint satisfiability, participants used Claripy, angr-management's solver engine, which includes a SAT solver. Claripy allows users to provide a constraint and produces an input that satisfies the constraint if one exists. Participants then re-executed the program to verify the vulnerability could be triggered. In this experiment, an input that produced a successful exploit caused the program to unload a module that should not be unloaded, triggering a use-after-free vulnerability, which could cause the program to crash.

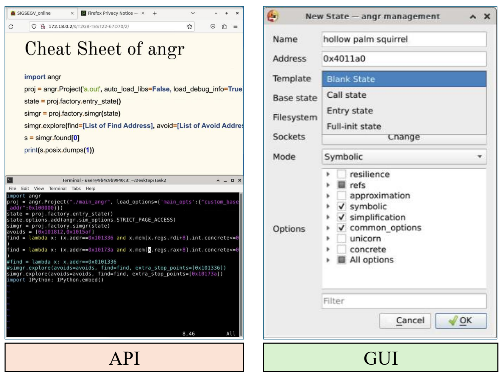
6 Impact of Features on Workflow (RQ2)

Given the SE workflow we identified in the prior section, we now turn to how participants actually completed this workflow in detail, specifically discussing differences between API and GUI participants' workflows. Figure 3 shows a summary of participant SE workflows divided between API participants (left) and API participants (right). We observed differences in each of the three phases regarding the ordering of steps and how steps were executed. We discuss each phase in turn, how each tools' interaction features impacted these differences, and how these differences influenced vulnerability discovery performance. As we discuss each tool-specific workflow, we note that every participant using the discussed tool completed all the steps mentioned unless specified with an indication of the number of participants (i.e., "n=X").

6.1 Setting up SE

In order to set up SE, users had to (1) comprehend the program, (2) instrument the code, and optionally (3) configure path exploration optimizations to set up SE. Although both API and GUI participants performed these steps, we observed differences in the order and manner in which they performed these steps, which impacted their performance.

Ordering of Steps. When participants used the API-based interface to set up SE, they first had to comprehend the program by reading through its functions, structure, and logic. Next, they prepared their initial SE script including the API function calls to create SE states and configure the necessary options (Figure 5a, API). Lastly, participants then optionally optimized this script for particular locations they wished to target or avoid via additional API calls. During this process, API participants had to write the SE script first, and then go back and forth between the target


SIGSEGV_online
Firefox Privacy Notice —
172.18.0.2/s/T2GB-TEST22-67D70/2/

Cheat Sheet of angr

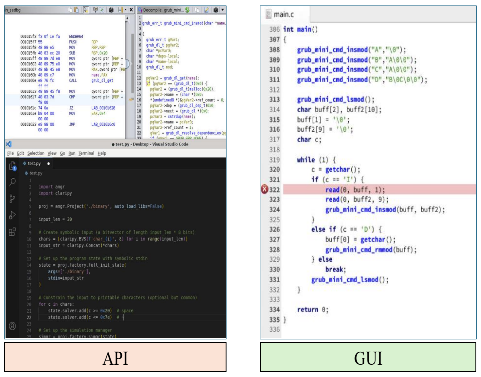
```
import angr
proj = angr.Project('a.out', auto_load_libs=False, load_debug_info=True
state = proj.factory.entry_state()
simgr = proj.factory.simgr(state)
simgr.explore(find=[List of Find Address], avoid=[List of Avoid Addres
s = simgr.found[0]
print(s.posix.dumps(1))
```

Terminal - user@9b4c9b9940c3: ~/Desktop/Task2
File Edit View Terminal Tabs Help

```
import angr
proj = angr.Project("./main_angr", load_options={'main_opts':{"custom_base
_addr":0x100000}})
state = proj.factory.entry_state()
state.options.add(angr.sim_options.STRICT_PAGE_ACCESS)
simgr = proj.factory.simgr(state)
avoids = [0x101812,0x1015af]
find = lambda x: (x.addr==0x101336 and x.mem[x.regs.rdi+8].int.concrete<=0
)
find = lambda x: (x.addr==0x10173a and x.mem[x.regs.rax+8].int.concrete<=0
)
#find = lambda x: x.addr==0x0101336
#simgr.explore(avoids=avoids, find=find, extra_stop_points=[0x101336])
simgr.explore(avoids=avoids, find=find, extra_stop_points=[0x10173a])
import IPython; IPython.embed()
                                                        8,46          All
```

Name: hollow palm squirrel
Address: 0x4011a0
Template: Blank State / Call state / Entry state / Full-init state
Base state
Filesystem
Sockets: Change
Mode: Symbolic
Options: ☐ resilience, ▨ refs, ☐ approximation, ✓ symbolic, ✓ simplification, ✓ common_options, ☐ unicorn, ☐ concrete, ▨ All options

(a) Initial State Configuration (F2).



```
001015f3 f3 0f 1e fa    ENDBR64
001015f7 55             PUSH     RBP
001015f8 48 89 e5       MOV      RBP,RSP
001015fb 48 83 ec 20    SUB      RSP,0x20
001015ff 48 89 7d e8    MOV      qword ptr [RBP +
00101603 48 89 75 e0    MOV      qword ptr [RBP +
00101607 48 8b 45 e8    MOV      RAX,qword ptr [RB
0010160b 48 89 c7       MOV      name,RAX
0010160e e8 76 fc       CALL     grub_dl_get
         ff ff
00101613 48 89 45 f8    MOV      qword ptr [RBP +
00101617 48 83 7d       CMP      qword ptr [RBP +
         f8 00
0010161c 74 0a          JZ       LAB_00101628
0010161e b8 04 00       MOV      EAX,0x4
         00 00
00101623 e9 98 00       JMP      LAB_001016c0
         00 00
```

```
{
  grub_err_t gVar1;
  grub_dl_t pgVar2;
  char *pcVar3;
  char *deps-local;
  char *name-local;
  grub_dl_t mod;

  pgVar2 = grub_dl_get(name);
  if (pgVar2 == (grub_dl_t)0x0) {
    pgVar2 = (grub_dl_t)malloc(0x20);
    pgVar2->name = (char *)0x0;
    *(undefined8 *)&pgVar2->ref_count = 0;
    pgVar2->dep = (grub_dl_dep_t)0x0;
    pgVar2->next = (grub_dl *)0x0;
    pcVar3 = xstrdup(name);
    pgVar2->name = pcVar3;
    pgVar2->ref_count = 1;
    gVar1 = grub_dl_resolve_dependencies(p
```

test.py - Desktop - Visual Studio Code
File Edit Selection View Go Run Terminal Help
test.py

```

import angr
import claripy

proj = angr.Project('./binary', auto_load_libs=False)

input_len = 20

# Create symbolic input (a bitvector of length input_len * 8 bits)
chars = [claripy.BVS(f'char_{i}', 8) for i in range(input_len)]
input_str = claripy.Concat(*chars)

# Set up the program state with symbolic stdin
state = proj.factory.full_init_state(
    args=['./binary'],
    stdin=input_str
)

# Constrain the input to printable characters (optional but common)
for c in chars:
    state.solver.add(c >= 0x20)  # space
    state.solver.add(c <= 0x7e)  # ~

# Set up the simulation manager
simgr = proj.factory.simgr(state)
```

```
int main()
{
    grub_mini_cmd_insmod("A","\0");
    grub_mini_cmd_insmod("B","A\0\0");
    grub_mini_cmd_insmod("C","A\0\0");
    grub_mini_cmd_insmod("D","B\0C\0\0");

    grub_mini_cmd_lsmod();
    char buff[2], buff2[10];
    buff[1] = '\0';
    buff2[9] = '\0';
    char c;

    while (1) {
        c = getchar();
        if (c == 'I') {
            read(0, buff, 1);
            read(0, buff2, 9);
            grub_mini_cmd_insmod(buff, buff2);
        }
        else if (c == 'D') {
            buff[0] = getchar();
            grub_mini_cmd_rmmod(buff);
        } else
            break;
        grub_mini_cmd_lsmod();
    }

    return 0;
}
```

(b) Avoid/Targeted Instruction Specification (F3).

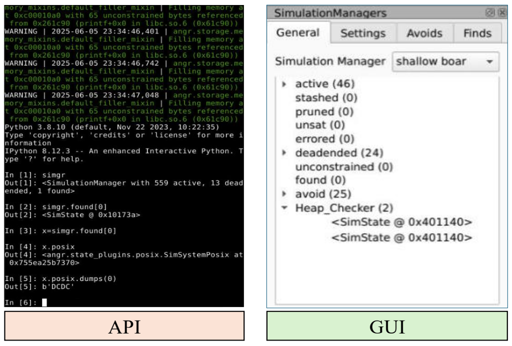
Fig. 5. **Initial State Configuration and Instruction Avoidance Interfaces** — We present the tool API- and GUI interfaces for Initial State Configuration feature (F2) and Avoid/Targeted Instruction Specification feature (F3).

program's code and the SE script, reading the program, identifying the optimizing location, and adding the location to the SE script repeatedly (Figure 5b, API). Such a process can be challenging for users, as they are regularly and rapidly switching windows, increasing user memory load [72]. API user A4 described this difficulty with the API tool, saying *"It is very powerful, but hard to set up."*

In contrast, while GUI participants also began by reading the program to comprehend how it is working, the rest of the phase was conducted differently. GUI participants did not need to prepare an external script because all the tools to set up SE (i.e., the *Avoid/Targeted Instruction Specification* and *Initial State Configuration* features) are included in the GUI. Instead, we observed that GUI participants performed path prioritization while reading the program for comprehension, quickly iterating between the two. The *Avoid/Targeted Instruction Specification* feature links the source code or disassembly with the path selection actions (Figure 5b, GUI), allowing users to prioritize paths while they are reading the program. In this way, GUI participants did not switch back and forth between script editing and other tasks allowing participants to help narrow their focus and reduce cognitive burdens: *"Using the GUI, I only needed to understand what was important to my target path of execution"* (G11).

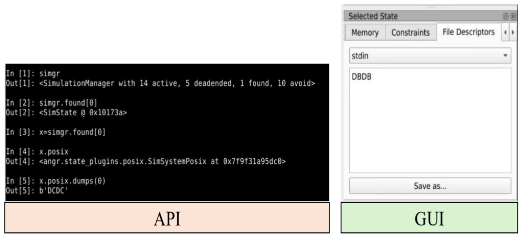
Once GUI participants finished comprehending the program and prioritizing paths, then they configured the initial state of the SE process using the *Initial State Configuration* feature (Figure 5a, GUI). This feature mimics the initialization code included by API participants at the start of their script allowing users to customize the state name and starting address, modify state templates and sockets, and select options, such as whether to use the Unicorn engine for state concretization. Because this initialization is done inside the GUI, it eliminates the need for users to write scripts manually.

Task Performance. We observed that the GUI features allowed users to more quickly set up their initial python script for SE. API participants spent 267.33 seconds on average to draft their initial python script (SD = 207.51, n = 9) whereas GUI participants only took an average of 4.85 seconds by interacting with the state setup menu (SD = 2.52, n = 10). This is due to the fact that unlike the API, the GUI-interface provides an initial SE state for its users, drastically reducing the amount of necessary manual work.


```
mory_mixins.default_filler_mixin | Filling memory a
t 0xc00010a0 with 65 unconstrained bytes referenced
 from 0x261c90 (printf+0x0 in libc.so.6 (0x61c90))
WARNING | 2025-06-05 23:34:46,401 | angr.storage.me
mory_mixins.default_filler_mixin | Filling memory a
t 0xc00010a0 with 65 unconstrained bytes referenced
 from 0x261c90 (printf+0x0 in libc.so.6 (0x61c90))
WARNING | 2025-06-05 23:34:46,742 | angr.storage.me
mory_mixins.default_filler_mixin | Filling memory a
t 0xc00010a0 with 65 unconstrained bytes referenced
 from 0x261c90 (printf+0x0 in libc.so.6 (0x61c90))
WARNING | 2025-06-05 23:34:47,048 | angr.storage.me
mory_mixins.default_filler_mixin | Filling memory a
t 0xc00010a0 with 65 unconstrained bytes referenced
 from 0x261c90 (printf+0x0 in libc.so.6 (0x61c90))
Python 3.8.10 (default, Nov 22 2023, 10:22:35)
Type 'copyright', 'credits' or 'license' for more i
nformation
IPython 8.12.3 -- An enhanced Interactive Python. T
ype '?' for help.

In [1]: simgr
Out[1]: <SimulationManager with 559 active, 13 dead
ended, 1 found>

In [2]: simgr.found[0]
Out[2]: <SimState @ 0x10173a>

In [3]: x=simgr.found[0]

In [4]: x.posix
Out[4]: <angr.state_plugins.posix.SimSystemPosix at
 0x755ea25b7370>

In [5]: x.posix.dumps(0)
Out[5]: b'DCDC'

In [6]: 
```

```
SimulationManagers
General   Settings   Avoids   Finds
Simulation Manager  shallow boar
▸ active (46)
  stashed (0)
  pruned (0)
  unsat (0)
  errored (0)
▸ deadended (24)
  unconstrained (0)
  found (0)
▸ avoid (25)
▾ Heap_Checker (2)
      <SimState @ 0x401140>
      <SimState @ 0x401140>
```

(a) Process Visualization (F4).


```
In [1]: simgr
Out[1]: <SimulationManager with 14 active, 5 deadended, 1 found, 10 avoid>

In [2]: simgr.found[0]
Out[2]: <SimState @ 0x10173a>

In [3]: x=simgr.found[0]

In [4]: x.posix
Out[4]: <angr.state_plugins.posix.SimSystemPosix at 0x7f9f31a95dc0>

In [5]: x.posix.dumps(0)
Out[5]: b'DCDC'
```

```
Selected State
Memory   Constraints   File Descriptors
stdin
DBDB
Save as...
```

(b) Symbolic State Investigation (F5).

Fig. 6. **Process and Symbolic State Interfaces** — We present the tool features in Process Visualization feature (F4) and Symbolic State Investigation feature (F5).

Interestingly, when investigating how quickly participants completed these steps, we did not observe a clear difference between API and GUI participants in the time spent in the setup phase with API participants spending an average of 199.33 seconds in this phase (SD = 89.72, n = 3) and GUI participants spending 127.56 seconds (SD = 82.28, n = 9). However, these similar times appear to be due to the fact that GUI participants performed more path prioritizations (3.56 locations prioritized on average) than API participants (1.33 locations on average), but they were able to complete them faster (35.83 seconds per location prioritization on average) than API participants (149.87 seconds per location prioritization on average). This suggests that while users spend similar amount of time setting up SE; since the GUI's path prioritization operation interface is more cost-efficient to users (less context switching between steps), users can conduct more operations within the similar time.

Note, while we allowed all participants to use online resources throughout the experiment and counted that time toward their total completion time, we found that only a few of the API participants (n=3) actually did so. All of them searched for angr documentation or community resources, which helped them draft their angr scripts.

6.2 Adjusting Path Exploration Priority

After setting up their initial SE script, users adjust the SE's exploration iteratively until suspicious symbolic states emerge; during this phase, users must balance the tension of ensuring important paths are explored, while minimizing path explosion and resource usage. To adequately explore relevant program states efficiently, users follow a number of steps (Figure 3), where they (1) run the symbolic execution exploration process, (2) observe the current SE progress, (3) stop the simulation if a suspicious state is found, or if the current exploration seems problematic (e.g., a path explosion occurs) or if suspicious states are not found, they may (4) adjust the path prioritization strategies, and run the SE process repeatedly. While both API and GUI participants follow these steps, we find that GUI participants can also adjust the SE process without restarting SE (as represented by the additional loop in Figure 3.GUI). This ability to change the analysis on the fly means GUI users spend less time per iteration of path prioritization then SE process execution.

Starting/Stopping/Pausing/Resuming Execution. To start and stop symbolic execution, API participants had to write one or more lines of code to invoke the SE APIs (Figure 4.API shows the command used to run their script and start the simulation) and then use keyboard interaction (e.g., CTRL-C) to stop it. If the users want to check the exploration process of SE, they need to pause and resume during the simulation, which is even more complicated; it requires users to set breakpoints in their scripts and interact with Python's debugging shell (e.g., IPython). As a result, users would be forced to switch between different interfaces, which may degrade performance [72]. In contrast, GUI participants can easily control an SE process by clicking the buttons provided by the *Process Control* feature. Figure 4.GUI shows the toolbar of controls provided by this feature which offer the ability to, from left to right, reverse execution, reverse one step in the simulation, continue the simulation, continue one step, or pause (stop if clicked twice) the simulation.

Notably, none of the API participants resumed a simulation, whereas over half of the GUI participants (n=7) paused and resumed simulations while searching for vulnerabilities. This difference is reflected in Figure 3 as an extra edge and steps from "adjusting path priority" to "continue simulation" and "inspect current SE progress." The key reason is that the GUI interface implements the resume feature as a simple button that users can easily click (the arrow pointing right in Figure 4.GUI). These observations suggest that the GUI design makes such operations considerably easier. As A9, an experience SE user, explained, *"I don't know how to tell angr to [continue to] advance."*

Inspect Progress. API participants have two main ways to monitor the current progress: (1) by viewing the script's output, if they inserted print statements, and (2) by stopping the execution and interacting directly with an interactive shell to investigate the SE process' logs. For example, Figure 6a.API shows a print statement (*simgr.found[0].posix.dump(0)*) that outputs information about the first stated identified through the SE simulation.

In contrast, GUI participants leverage the *Progress Visualization* feature by checking the stash status located on the right-hand side of the interface (Figure 6a.GUI). This feature gives participants a summary of the SE process' progress, bucketing identified states into one of several categories, referred to as "stashed." These included states currently being explored, those avoided based on path prioritization, and those found to be suspicious. GUI users could then obtain more detailed information about a specific symbolic state, such as register lists, file descriptors, and memory locations, using the *Symbolic State Investigation* feature without pausing or stopping the simulation (Figure 6b.GUI). As noted by G2, these designs readily assist experts in actually making progress during SE, *"The GUI makes it much easier to perform symbolic execution and inspect the SE progress compared to manual programming"*

Adjust Path Priority. When adjusting path priority, API participants first had to stop SE process execution, identify the target location by analyzing the source code. They then map the source code to the disassembled code in order to retrieve the corresponding address, which they manually add to their script and re-run the SE process. If they want to verify addresses already present in the script, they have to remember the address, search for it in the disassembly, and switch back and forth between the script, source code, and disassembled view to complete the task (Figure 5b.API). In contrast, the GUI's *Avoid/Targeted Instruction Specification* feature allows GUI participants to adjust their decisions by adding or removing addresses while reading the source code or the disassembly code and clicking the line of code to be prioritized, which toggles prioritization on or off (Figure 5b.GUI). Importantly, this can be done mid-SE process execution GUI participants were able to pause the simulation, adjust path prioritization, and resume the simulation without needing to re-run earlier parts of the simulation. It is possible this feature would not be used as it could be more complicated to consider an in-progress execution when making prioritization decisions. However, that was not the case in practice, as all our GUI participants took advantage of this feature to allow more efficient path prioritization.

Task Performance. Although the methods used by API and GUI participants to inspect the current symbolic execution progress were quite different, the time they spent to inspect the SE process' progress appeared to be similar. API participants spent an average of 9.6 seconds observing the current progress through script output (SD = 2.26, n = 2). They also spent 77.25 seconds interacting with the shell to obtain more detailed information about the ongoing simulation (SD = 119.64, n = 5). In comparison, GUI participants spent an average of 69.48 seconds interacting with the simulation manager stash to monitor and explore the current SE progress (SD = 35.14, n = 8). As an our interfaces does not have dramatic impact on time, this may suggest that time is mostly spent on information comprehension rather than simply viewing the variables which are displayed in our interface.

In the path prioritization step, while API and GUI participants spent a comparable overall time adjusting path priorities (API: M = 50.14s, SD = 36.72, n = 6; GUI: M = 47.51s, SD = 37.37, n = 8), their underlying behavior differed. GUI participants performed more operations during this step, with an average of 3.23 path prioritizations compared to 2.23 by API participants. As a result, the average time per prioritization was lower for GUI participants, at 14.71 seconds, while API participants took 22.48 seconds per prioritization. This is similar to the relationship we observed for path prioritization during the initial setup phase (Section 6.1), while the GUI did not speed up the process overall, more actions were conducted in the same time span. This implies that GUI support contributes to improving the efficiency of path adjustment during symbolic execution.

6.3 Analyzing Suspicious Symbolic States

After stopping the SE process or once the process had exhaustively searched all program states, the API participants could use the interactive shell to inspect states logged as potentially suspicious during the simulation (Figure 6b.API). This step was performed in the same way as the "inspect progress" step in the prior phase and took 77 seconds on average to complete (SD = N/A, n = 1).

In contrast, GUI participants used the *Progress Visualization* (Figure 6a.GUI) and *Symbolic State Investigation* (Figure 6b.GUI) features, as with the "inspect progress" step to review the final state after execution was stopped. With these features, participants looked through states identified states listed in the "found" and "Heap_Checker" interfaces. For each state, participants used the *Symbolic State Investigation* feature to extract the symbolic inputs and determine whether there was an satisfiable input that could exploit the vulnerability. GUI participants were slightly faster completing this step as they spent 53.75 seconds on average completing this step (SD = 51.68, n = 4) and noticably commented on it's usability, *"Checking the inputs is also convenient"* (G7) and *"Knowing memory access violations and easily accessing the input which led to them is incredibly useful and was very easy to use"* (G11).

6.4 Overall Performance

In this section, we compare the overall performance and participants between assigned tools, specifically considering their task success, participants ability to effectively prioritize paths to drive the tool toward the vulnerability, their reasons for giving up on using the tool, and SUS scores.

Vulnerability Correctness. There were three possible outcomes for each participant's task: 1) used SE to correctly identify the vulnerability (*Correct Answer*), 2) used SE but did not identify, or incorrectly identified, the vulnerability (*Wrong Answer*), or 3) quit the task, or used another tool (*Gave-up*).

In our results, we find that of the 13 participants who used API-based symbolic execution, only a single participant was able to correctly find the vulnerability while all the others either yield a wrong answer or gave up. In contrast

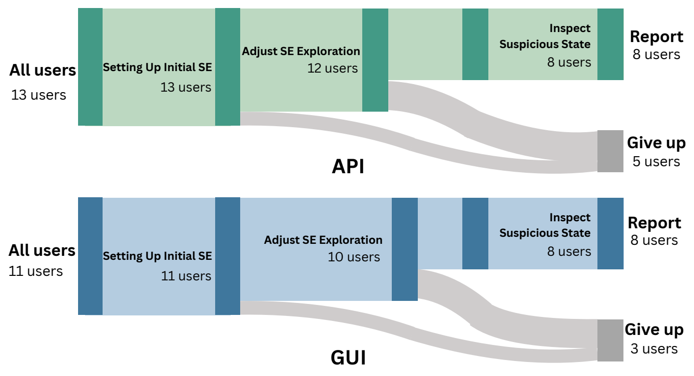


Fig. 7. **Outcomes of API- and GUI-based Participants** — We present the number of participants that complete, or give up, at various parts of SE workflow.

of the 11 participants who used GUI-based symbolic execution, 4 were able to correctly find the vulnerability. This suggests that GUI-participants may be more likely to correctly use SE to identify vulnerabilities. We did not observe any clear pattern between participants' success and their prior experience. Additionally, we also find that GUI participants completed the task in a shorter time. Among those who successfully found the vulnerability, GUI participants took 73 minutes on average (SD = 0.04, n = 4), while API participant spent 238 minutes to find it (SD = N/A, n = 1). However, when considering all participants regardless of success, the overall time spent by the two groups is similar: API participants spent an average of 84 minutes (SD = 0.38, n = 13), while GUI participants spent 86 minutes (SD = 0.39, n = 11) over the whole experiment.

Path Prioritization Ineffectiveness. At a high level, we define a path prioritization operation—to prioritize or avoid a program location—to be *effective* if the prioritization does not prevent the exploration of the path containing the vulnerability. If the prioritization *excludes* the path containing the vulnerability, then it would be *ineffective*. To this end, we evaluate each path prioritization operation by running SE with and without the prioritization to see if the vulnerability can still be identified with the prioritization. We observed that the effectiveness rate—the proportion of effective path-optimization operations—was similar for GUI and API participants. GUI participants made 46 total efficient path prioritization out of 104 total prioritization (44%), while API participants made 18 total efficient path prioritizations out of 34 total paths (53%). This corresponds to an average of 4.6 (SD = 1.78) efficient paths out of 10.4 total paths (SD = 5.04) per GUI participant, and 2.57 efficient paths (SD = 1.99) out of 4.86 total paths (SD = 3.58) of API participants. We observe that the GUI does not clearly enhance participants' decision-making intuition or lead to more effective prioritization choices. We hypothesize that the GUI interface outperforms the API in vulnerability discovery due to the increased number of path optimization operations it encourages. However, the effectiveness of each individual operation is not improved by the GUI. Nevertheless, because GUI participants perform more operations overall, the cumulative impact on mitigating path explosion is greater, ultimately resulting in better vulnerability discovery performance.

Reasons for Giving Up. We further investigate the cases where users gave up on using SE tools. Specifically, we look at the phase at which users gave up, and their reasons for doing so. Figure 7 shows the statistics of the gave-up stage for both API and GUI participants.

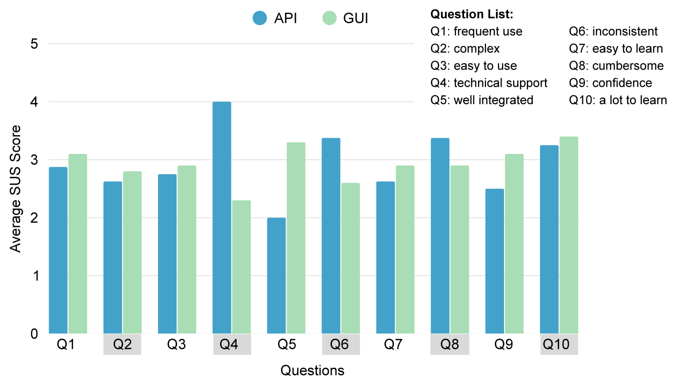


Fig. 8. **SUS Scores of API- and GUI-based Tools** — We present the average SUS score for all 10 questions seperated by whether they are assigned the API-based tool (left bar) or whether they are assigned the GUI-based tool (right bar). SUS questions are included in the supplementary materials [6].

Initializing SE: One participant from API group gave up during the setup of the initial symbolic execution pass. They explained that drafting an initial SE script was a heavy burden, and the reason for them giving up, *"After that I spent 30–40 minutes trying to write an angr script to make it discover the input ...unfortunately this takes too much time and I gave up"* (A7). We observed that the user attempted to statically identify multiple avoided program locations in the first pass. Unfortunately, identifying and configuring these program locations can be particularly effort-consuming for API participants due to the lack of integrated interface for SE configuration and program analysis. One user of GUI-based symbolic execution also gave up before executing any code, citing difficulty in locating the interface to begin symbolic execution: *"[I] cannot find the symbolic execution's start button."* (G10). Thus GUI's while seemingly more simple, it may also introduce UI design that can similarly detract from usability.

Adjusting SE: Four API and two GUI participants gave up during the SE adjustment phase. Notably, for three of the API users, and two of the GUI, they were unable find the vulnerability due to inherent difficulties in properly setting SE path prioritizations; for instance, by avoiding program locations that were critical for reaching the vulnerability. Notably; however, two API participant gave up because of the complexity of customizing and checking the program state and navigating multiple library function, *"I failed to find how to check for non-crashing UAF in angr."* (A7). This may imply that while interface improvements can help prevent some failures, difficulties inherent to SE operation, and not the interface, still cause many failures.

Participant perception of tool usability. The average SUS score for angr was 40 (SD = 11.91, n = 13), and 53 for angr-management (SD = 17.12, n = 11), which correspond to an F and D grade respectively according to Lewis' grading scale [47]. From the SUS grading, we can tell that although angr-management provided better user experience than angr, both still have a long way to go before becoming well-designed software. The average item scores for both groups are presented in Figure 8. From the figure, we observe that angr-management performs better than angr on all positive-worded items. The negative-worded items, however, reveal more interesting differences. Angr-management receives higher scores on items related to unnecessary complexity and having a long learning curve. In contrast, angr receives higher scores on items indicating the need for technical support, system inconsistency, and a cumbersome usage experience. Notably, the largest gap between angr and angr-management appears on the item regarding the need for a technical person (on average 4.0 vs. 2.3), suggesting that the GUI design indeed makes the tool easier to use. However, the GUI design still requires further improvement to become clearer and more intuitive.

In addition, we also investigate users' confidence in using symbolic execution tools based on the participants' answer for this question. While we do find that users who succeeded in discovering vulnerabilities reported greater confidence in using their tool compared to those who failed in the task (on average, 4.0 vs. 2.5) we notice that this difference existing, but is smaller between GUI-based symbolic execution users and the API-based symbolic execution participant (on average, 3.1 vs. 2.5).

7 Discussion and Recommendations

Our results demonstrate GUI-based symbolic execution's feasibility and potential to advance the state-of-the-art in vulnerability discovery capabilities, specifically showing that a GUI-based interface can reduce friction in the SE process, allowing users to more quickly iterate and identify vulnerabilities. To better understand the broader significance of these findings, we situate our results within prior human-factors research in vulnerability discovery.

7.1 Interpreting Our Findings Through Prior Human-Factors Research

Prior work across vulnerability discovery and reverse engineering has shown that analyst performance is strongly shaped by tool usability rather than analytic ability alone. Studies of fuzzing and static-analysis tool use highlight how high setup costs create early barriers to effective analysis [61], while research on reverse-engineering workflows show that uninterrupted transitions between comprehension, probing, and observation enable more effective analysis [13, 82]. Other empirical studies of professional vulnerability discovery show that analysts depend on tightly integrated workflows spanning static analysis, tool outputs, and runtime interaction in order to sustain sense-making during complex attacks [20]. Our study findings align closely with these observations. GUI-based symbolic execution reduced context switching and streamlined iterative exploration, allowing participants to perform more frequent path prioritization operations, inspect symbolic states with lower overhead, and revise exploration strategies without interrupting execution.

Prior work on understanding the minds of reverse engineers shows that expert analysts outperform novices not by reading code faster, but by more strategically managing their exploration and rapidly dismissing irrelevant paths. The RE-Mind study found that experts skip entire branches more frequently after a single brief inspection, and reverse substantially less overall code despite similar first-pass reading times [51]. Our findings align with this conclusion: while GUI-based symbolic execution did not significantly reduce total task duration, it enabled greater iteration frequency and higher vulnerability discovery success. This suggests that in complex vulnerability discovery workflows, usability gains are used to enable deeper exploratory analysis rather than to finish more quickly.

Importantly, our study extends prior usability research into an under-examined domain: symbolic execution. Although symbolic execution has been widely studied from an algorithmic perspective focusing on solver optimization, path-selection heuristics, and scalability, its usability and human factors aspects have received little direct empirical attention. We provide the first controlled study evaluating how interface design affects the workflows and performance of expert symbolic-execution practitioners. Our findings, therefore, position GUI-based symbolic execution not merely as a convenience feature but as an essential usability evolution to translate symbolic execution's analytical power into effective real-world vulnerability discovery practice.

However, this is only a first step. Our study results also suggest several potential avenues to improve GUI-based symbolic execution, both immediately by SE developers and through future research. In this section, we discuss recommendations for each group in turn.

7.2 Recommendations for SE developers

Visualize Program Information To Help Users Make More Effective Path Optimization Decisions. We did not observe an improvement in the effectiveness rate of path optimization operations from the GUI-based symbolic execution and most participants using this tool still made path optimization decisions that lead to unuseful or misleading path execution. Therefore, a key weakness in the current tool appears to be in its lack of ability to help users better identify which paths are most likely to be fruitful. Future GUI-based symbolic execution should investigate how to visualize relevant program information to help users to make optimization decisions. As suggested by G4, GUI-based symbolic execution could *"give recommendations for avoid/find addresses."* This can be achieved by incorporating existing path optimization techniques[66, 88, 89, 91] and visualizing their generated results within the GUI.

Provide Information About Detected Vulnerable Symbolic Execution States To Support Triage. While GUI-based SE presents detected vulnerable symbolic execution states, it does not provide information regarding what this state indicates about the vulnerability. Multiple participants raised this as a suggested area of improvement, specifically that GUI-based symbolic execution should provide an explanation of the vulnerable state. Both G9 and G4 suggested including information to "help in triaging the bug" and show the type of vulnerability associated with the detected state.

Provide Details Explaining Symbolic States. Multiple participants also recommended including visualizations that explain the symbolic execution process in more detail. This is necessary to help users see under-the-hood of the analysis and understand its constraints as they debug and iterate on the configuration. For example, G11 was confused about "how or if invalid memory accesses on the stack are tracked or communicated." At a minimum, this suggests the need for a more robust mechanism for introspecting the current symbolic execution process state than is currently provide by GUI-based SE.

7.3 Recommendation For Researchers

Intentions vs. Behaviors. We conducted an observational experiment in this paper. As the first study to examine the workflow, effectiveness, and efficiency of symbolic execution in vulnerability discovery, our primary goal is to assess whether the recent GUI-based symbolic execution outperforms API-based symbolic execution without requiring an understanding of user intentions. A promising direction for future research is to analyze the intentions of domain experts when making decisions throughout the process as this might reveal misconceptions causing inefficiencies or places where the GUI is unable to match users goals. Gaining insight into these intentions could guide the design of GUI-based symbolic execution, enhancing its effectiveness and efficiency. We leave this exploration for future work.

Feature Ablation Tests. In this paper, we focus on the overall design of GUI-based symbolic execution for vulnerability discovery tasks. Since GUI-based symbolic execution combines multiple features — such as state prioritization and state statistics and inspection, a valuable direction for future research would be to evaluate the individual impact of each feature on effectiveness and efficiency. For instance, one could disable one feature while retaining the other to observe participant behavior and performance with a single feature. This approach would help clarify the specific contributions of each feature in the GUI-based symbolic execution design.

GUI vs. CLI. In our study, we found that debugging symbolic execution enhances user performance in vulnerability discovery. Building on this insight, future research could explore the effectiveness of graphical visualization in symbolic execution and vulnerability discovery. For example, one could compare GUI-based symbolic execution with a Command Line Interface (CLI)-based symbolic execution interface, such as a symbolic execution debugger analogous to `pwndbg` for `gdb`. Implementing a symbolic execution plugin for a Python debugger to display symbolic execution-related information would enable users to operate in a command-line interface, providing an alternative to a graphical interface.

Vulnerability Discovery vs. Other Applications. Besides vulnerability discovery, another potential research direction would be to examine symbolic execution's performance in other applications, such as software testing and reverse engineering. It would be valuable to investigate whether GUI-based symbolic execution maintains its advantages in these contexts and to understand the workflows involved in applying symbolic execution to diverse applications.

8 Conclusion

This paper evaluates the impact GUIs have on the workflow and performance of expert-driven symbolic execution. Through a controlled experiment where 24 SE practitioners were assigned either an API- or GUI-based interface and asked to complete a vulnerability discovery task, we identified differences in the workflows and performance between each group. Compared to API-based workflows, the GUI-based interface reduced context switching and restarts, allowing participants to more rapidly iterate through possible configurations and path optimizations. Consequently, participants assigned to the GUI-based interface performed more path operations, were more likely to successfully discover the vulnerability, and did so more quickly. As a whole, our study demonstrates that GUI-based interfaces have the potential to enhance the effectiveness of SE-based vulnerability discovery.

References

[1] 2022. How I gave ManticoreUI a makeover. https://blog.trailofbits.com/2022/12/15/manitcoreui-symbolic-execution-gui/ Section: posts.

[2] 2022. Manticore GUIs made easy. https://blog.trailofbits.com/2022/12/13/manticore-gui-plugin-binary-ninja-ghidra/ Section: posts.

[3] 2025. Pwn2Own: The World's Most Elite Hacking Competition. https://www.trendmicro.com/en_us/zero-day-initiative/pwn2own.html

[4] Omer Akgul, Taha Eghtesad, Amit Elazari, Omprakash Gnawali, Jens Grossklags, Michelle L. Mazurek, Daniel Votipka, and Aron Laszka. 2023. Bug Hunters' Perspectives on the Challenges and Benefits of the Bug Bounty Ecosystem. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 2275–2291. https://www.usenix.org/conference/usenixsecurity23/presentation/akgul

[5] angr Project Developers. 2025. angr Documentation — Symbolic execution. https://docs.angr.io/en/latest/core-concepts/symbolic.html

[6] Anonymous Authors. 2025. I Can SE Clearly Now: Investigating the Effectiveness of GUI-based Symbolic Execution for Software Vulnerability Discovery - Supplemental Materials. https://anonymous.4open.science/r/chi2026-SupplementaryMaterials-C0E9.

[7] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. 2014. Automatic exploit generation. *Commun. ACM* 57, 2 (2014), 74–84.

[8] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.

[9] Gabriel Bessler, Josh Cordova, Shaheen Cullen-Baratloo, Sofiane Dissem, Emily Lu, Sofia Devin, Ibrahim Abughararh, and Lucas Bang. 2021. Metrinome: Path Complexity Predicts Symbolic Execution Path Explosion. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 29–32. https://doi.org/10.1109/ICSE-Companion52605.2021.00028 ISSN: 2574-1926.

[10] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. 2022. SENinja: A symbolic execution plugin for Binary Ninja. *SoftwareX* 20 (2022), 101219. https://doi.org/10.1016/j.softx.2022.101219

[11] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECT—a formal system for testing and debugging programs by symbolic execution. *SIGPLAN Not.* 10, 6 (apr 1975), 234–245. https://doi.org/10.1145/390016.808445

[12] John Brooke et al. 1996. SUS-A quick and dirty usability scale. *Usability evaluation in industry* 189, 194 (1996), 4–7.

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

[14] BugCrowd. 2018. 2019 Inside the Mind of a Hacker Report Reveals Gender Imbalance, Hacker Education and Highest Paid Crowds. https://www.bugcrowd.com/blog/2019-inside-the-mind-of-a-hacker-report-reveals-gender-imbalance-hacker-education-and-highest-paid-crowds/

[15] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.

[16] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2008. EXE: Automatically Generating Inputs of Death. *ACM Trans. Inf. Syst. Secur.* 12, 2, Article 10 (dec 2008), 38 pages. https://doi.org/10.1145/1455518.1455522

[17] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. 2011. Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*. 1066–1071.

[18] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (Feb. 2013), 82–90. https://doi.org/10.1145/2408776.2408795

[19] Chen Cao, Le Guan, Jiang Ming, and Peng Liu. 2020. Device-agnostic firmware execution is possible: A concolic execution approach for peripheral emulation. In *Proceedings of the 36th Annual Computer Security Applications Conference*. 746–759.

[20] M. Ceccato, P. Tonella, C. Basile, B. Coppens, B. De Sutter, P. Falcarin, and M. Torchiano. 2017. How Professional Hackers Understand Protected Code while Performing Attack Tasks. In *IEEE International Conference on Program Comprehension*. IEEE, 154–164. https://doi.org/10.1109/ICPC.2017.2 arXiv:1704.02774

[21] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 380–394.

[22] Vitaly Chipounov and George Candea. 2010. Reverse engineering of binary device drivers with RevNIC. In *Proceedings of the 5th European Conference on Computer Systems* (Paris, France) *(EuroSys '10)*. Association for Computing Machinery, New York, NY, USA, 167–180. https://doi.org/10.1145/1755913.1755932

[23] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 265–278. https://doi.org/10.1145/1950365.1950396

[24] Stephen V Cole. 2022. Impact of capture the flag (CTF)-style vs. traditional exercises in an introductory computer security class. In *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1*. 470–476.

[25] Ubuntu Community. 2023. Grub2. Ubuntu Community Help Wiki. https://help.ubuntu.com/community/Grub2.

[26] Juliet Corbin and Anselm Strauss. 2014. *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Sage Publications.

[27] The MITRE Corporation. 2020. CVE-2020-25632. https://www.cve.org/CVERecord?id=CVE-2020-25632

[28] CVE Details. 2025. CVE Vulnerabilities by Types. https://www.cvedetails.com/vulnerabilities-by-types.php

[29] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. 2016. BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 653–656.

[30] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. 2013. {FIE} on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *22nd USENIX Security Symposium (USENIX Security 13)*. 463–478.

[31] Sriharsha Etigowni, Dave Tian, Grant Hernandez, Saman Zonouz, and Kevin Butler. 2016. CPAC: securing critical infrastructure with cyber-physical access control. In *Proceedings of the 32nd annual conference on computer security applications*. 139–152.

[32] 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 vulnerability discovery. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1997–2014.

[33] GAIT. 2025. Ghidra Angr Integration Tool - Symbolic Execution for serpentine monsters! https://foundryzero.co.uk/2024/08/23/ghidra-angr-integration-tool.html

[34] Xiang Gao, Shin Hwei Tan, Zhen Dong, and Abhik Roychoudhury. 2018. Android testing via synthetic symbolic execution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 419–429.

[35] Fabio Gritti, Lorenzo Fontana, Eric Gustafson, Fabio Pagani, Andrea Continella, Christopher Kruegel, and Giovanni Vigna. 2020. Symbion: Interleaving symbolic with concrete execution. In *2020 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 1–10.

[36] Hackerone. 2019. *2019 Hacker-Powered Security Report*. Technical Report. Hackerone, San Francisco, California. https://www.hackerone.com/resources/reporting/the-hacker-powered-security-report-2019

[37] Hackerone. 2025. *The Rise of the Bionic Hacker*. Technical Report. Hackerone, San Francisco, California. https://www.hackerone.com/resources/hacker-powered-security/9th-hpsr-human-advantage

[38] Reiner Hähnle, Marcus Baum, Richard Bubel, and Marcel Rothe. 2010. A visual interactive debugger based on symbolic execution. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*. 143–146.

[39] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to fuzz from symbolic execution with application to smart contracts. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*. 531–548.

[40] Jingxuan He, Gishor Sivanrupan, Petar Tsankov, and Martin Vechev. 2021. Learning to Explore Paths for Symbolic Execution. 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, 2526–2540. https://doi.org/10.1145/3460120.3484813

[41] Martin Hentschel, Richard Bubel, and Reiner Hähnle. 2014. Symbolic execution debugger (SED). In *Runtime Verification: 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings 5*. Springer, 255–262.

[42] Dávid Honfi, Andras Voros, and Zoltán Micskei. 2015. Seviz: A tool for visualizing symbolic execution. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–8.

[43] Jie Hu, Yue Duan, and Heng Yin. 2024. Marco: A Stochastic Asynchronous Concolic Explorer *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3597503.3623301

[44] Alberto Garcia Illera. 2025. The Ponce Project. https://github.com/illera88/Ponce original-date: 2016-07-02T03:48:56Z.

[45] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering (ICSE)*. 672–681. https://doi.org/10.1109/ICSE.2013.6606613

[46] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[47] James R. Lewis and Jeff Sauro. 2013. Item Benchmarks for the System Usability Scale (SUS). *UXPA Journal* (2013), 158–167. https://uxpajournal.org/item-benchmarks-system-usability-scale-sus/

[48] Penghui Li, Wei Meng, Mingxue Zhang, Chenlin Wang, and Changhua Luo. 2024. Holistic Concolic Execution for Dynamic Web Applications via Symbolic Interpreter Analysis. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 222–238.

[49] Shang-Wei Lin, Palina Tolmach, Ye Liu, and Yi Li. 2022. SolSEE: a source-level symbolic execution engine for solidity. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1687–1691. https://doi.org/10.1145/3540250.3558923

[50] I. Scott MacKenzie. 2013. *Human-Computer Interaction: An Empirical Research Perspective*. Morgan Kaufmann, San Francisco, CA. See Section 5.11: Order effects, counterbalancing, and Latin squares.

[51] 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

[52] James Mattei, Madeline McLaughlin, Samantha Katcher, and Daniel Votipka. 2022. A Qualitative Evaluation of Reverse Engineering Tool Usability. In *ACSAC 2022, Annual Computer Security Applications Conference*.

[53] Lucas McDaniel, Erik Talvi, and Brian Hay. 2016. Capture the flag as cyber security introduction. In *2016 49th hawaii international conference on system sciences (hicss)*. IEEE, 5479–5486.

[54] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. 2019. Reliability and Inter-rater Reliability in Qualitative Research: Norms and Guidelines for CSCW and HCI Practice. *Proc. ACM Hum.-Comput. Interact.* 3, CSCW, Article 72 (Nov. 2019), 23 pages. https://doi.org/10.1145/3359174

[55] Nariman Mirzaei, Sam Malek, Corina S Păsăreanu, Naeem Esfahani, and Riyadh Mahmood. 2012. Testing android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes* 37, 6 (2012), 1–5.

[56] Sudip Mittal, Prajit Kumar Das, Varish Mulwad, Anupam Joshi, and Tim Finin. 2016. Cybertwitter: Using twitter to generate alerts for cybersecurity threats and vulnerabilities. In *2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*. IEEE, 860–867.

[57] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1186–1189.

[58] David Molnar Patrice Godefroid, Michael Levin. 2008. Automated Whitebox Fuzz Testing. In *Network and Distributed System Security Symposium.*

[59] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 697–710.

[60] 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.

[61] Stephan Plöger, Mischa Meier, and Matthew Smith. 2023. A Usability Evaluation of AFL and libFuzzer with CS Students. In *Conference on Human Factors in Computing Systems.* https://doi.org/10.1145/3544548.3581178

[62] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with {SymCC}: Don't interpret, compile!. In *29th USENIX Security Symposium (USENIX Security 20)*. 181–198.

[63] Sebastian Poeplau and Aurélien Francillon. 2021. SymQEMU: Compilation-based symbolic execution for binaries. In *NDSS 2021, Network and Distributed System Security Symposium.* Internet Society.

[64] Weizhong Qiang, Yuehua Liao, Guozhong Sun, Laurence T Yang, Deqing Zou, and Hai Jin. 2017. Patch-related vulnerability detection based on symbolic execution. *IEEE Access* 5 (2017), 20777–20784.

[65] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. Vuzzer: Application-aware evolutionary fuzzing. In *2017 Network and Distributed System Security (NDSS) Symposium:[Proceedings]*. Internet Society, 1–14.

[66] Nicola Ruaro, Kyle Zeng, Lukas Dresel, Mario Polino, Tiffany Bao, Andrea Continella, Stefano Zanero, Christopher Kruegel, and Giovanni Vigna. 2021. SyML: Guiding Symbolic Execution Toward Vulnerable States Through Pattern Learning. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses* (San Sebastian, Spain) *(RAID '21)*. Association for Computing Machinery, New York, NY, USA, 456–468. https://doi.org/10.1145/3471621.3471865

[67] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A symbolic execution framework for javascript. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 513–528.

[68] Ben Shneiderman and Catherine Plaisant. 2010. *Designing the user interface: strategies for effective human-computer interaction.* Pearson Education India.

[69] Yan Shoshitaishvili, Ruoyu Wang, Audrey Dutcher, Lukas Dresel, Eric Gustafson, Nilo Redini, Paul Grosen, Colin Unger, Chris Salls, Nick Stephens, Christophe Hauser, John Grosen, Christopher Kruegel, and Giovanni Vigna. 2024. angr-management. https://github.com/angr/angr-management

[70] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey 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 *IEEE Symposium on Security and Privacy*.

[71] 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 *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 347–362. https://doi.org/10.1145/3133956.3134105 arXiv:arXiv:1708.02749v1

[72] Sidney L Smith and Jane N Mosier. 1986. *Guidelines for designing user interface software.* Technical Report. Mitre Corporation Bedford, MA.

[73] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. 2021. {SmarTest}: Effectively hunting vulnerable transaction sequences in smart contracts through language {Model-Guided} symbolic execution. In *30th USENIX Security Symposium (USENIX Security 21)*. 1361–1378.

[74] 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*. 1–16.

[75] Nikolai Tillmann and Peli de Halleux. 2008. Pex - White Box Test Generation for .NET. In *Proc. of Tests and Proofs (TAP'08)* (proc. of tests and proofs (tap'08) ed.) *(LNCS)*, Vol. 4966. Springer Verlag, 134–153. https://www.microsoft.com/en-us/research/publication/pex-white-box-test-generation-for-net/

[76] David Trabish, Shachar Itzhaky, and Noam Rinetzky. 2021. A bounded symbolic-size model for symbolic execution. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1190–1201.

[77] David Trabish, Timotej Kapus, Noam Rinetzky, and Cristian Cadar. 2020. Past-sensitive pointer analysis for symbolic execution. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 197–208.

[78] David Trabish and Noam Rinetzky. 2020. Relocatable addressing model for symbolic execution. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 51–62.

[79] TrailofBits. 2025. ManticoreUI. https://github.com/trailofbits/ManticoreUI original-date: 2021-07-07T21:15:05Z.

[80] TyeYeah. 2020. Learn Symbolic Execution and angr. https://tyeyeah.github.io/2020/03/05/2020-03-05-Learn-Symbolic-Execution-and-angr/

[81] Artem Voronkov, Leonardo A Martucci, and Stefan Lindskog. 2019. System Administrators Prefer Command Line Interfaces, Don't They? An Exploratory Study of Firewall Interfaces. In *Proceedings of the Fifteenth Symposium on Usable Privacy and Security*.

[82] Daniel Votipka, Kelsey R Fulton, James Parker, Matthew Hou, Michelle L Mazurek, and Michael Hicks. 2020. Understanding security mistakes developers make: Qualitative analysis from build it, break it, fix it. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 109–126.

[83] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S. Foster, and Michelle L. Mazurek. 2020. An Observational Investigation of Reverse Engineers' Processes. In *The Proceedings of the 29th USENIX Security Symposium*.

[84] Daniel Votipka, Rock Stevens, Elissa Redmiles, Jeremy Hu, and Michelle Mazurek. 2018. Hackers vs. testers: A comparison of software vulnerability discovery processes. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 374–391.

[85] Fish Wang and Yan Shoshitaishvili. 2017. Angr - The Next Generation of Binary Analysis. In *2017 IEEE Cybersecurity Development (SecDev)*. 8–9. https://doi.org/10.1109/SecDev.2017.14

[86] Carter Yagemann, Matthew Pruett, Simon P Chung, Kennon Bittick, Brendan Saltaformaggio, and Wenke Lee. 2021. {ARCUS}: symbolic root cause analysis of exploits in production systems. In *30th USENIX Security Symposium (USENIX Security 21)*. 1989–2006.

[87] 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

[88] Shuangjie Yao and Dongdong She. 2025. Empc: Effective Path Prioritization for Symbolic Execution with Path Cover . In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 2772–2790. https://doi.org/10.1109/SP61157.2025.00190

[89] Jaehan Yoon and Sooyoung Cha. 2024. FeatMaker: Automated Feature Engineering for Search Strategy of Symbolic Execution. 1, FSE, Article 108 (July 2024), 22 pages. https://doi.org/10.1145/3660815

[90] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*. 745–761.

[91] Shunfan Zhou, Zhemin Yang, Dan Qiao, Peng Liu, Min Yang, Zhe Wang, and Chenggang Wu. 2022. Ferry: State-Aware Symbolic Execution for Exploring State-Dependent Program Paths. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 4365–4382. https://www.usenix.org/conference/usenixsecurity22/presentation/zhou-shunfan