SoK: A Modularized Framework for Symbolic Execution and Application for Usable Tool Design

James Mattei
Tufts University
MA, USA

Andrew Lin
Tufts University
MA, USA

Jasper Geer
University of British Columbia
Canada

Jie Hu
Arizona State University
AZ, USA

Moritz Schloegel
CISPA
Germany

Tiffany Bao
Arizona State University
AZ, USA

Daniel Votipka
Tufts University
MA, USA

Abstract

Symbolic Execution (SE) is an important and foundational software testing technique that has grown and evolved in its use over the decades. Prior work has cataloged this evolution, but this paper seeks to identify opportunities to go beyond existing designs and push forward the boundaries of its use by breaking down critical components of SE and outlining current approaches to each. To this end, we performed a systemization of 225 SE papers from the last 15 years to identify common design patterns and use cases. From this review, we distill five distinct modules of the SE architecture and discuss current implementations for each. This division of SE into modules can highlight opportunities for future improvements to SE by helping research focus on individual components. To demonstrate the modules' utility, we use the modules to identify changes for each module necessary to improve SE usability building on a second systemization of 66 papers containing insights about tooling usability.

CCS Concepts

• **Security and privacy → Software security engineering**; *Usability in security and privacy*.

Keywords

Symbolic Execution, Usable Program Analysis

ACM Reference Format:
James Mattei, Andrew Lin, Jasper Geer, Jie Hu, Moritz Schloegel, Tiffany Bao, and Daniel Votipka. 2026. SoK: A Modularized Framework for Symbolic Execution and Application for Usable Tool Design. In *Proceedings of the 2026 ACM Secure Development Conference (SecDev '26), July 5–6, 2026, Montreal, QC, Canada.* ACM, New York, NY, USA, 21 pages. https://doi.org/10.1145/3805773.3805996

This work is licensed under a Creative Commons Attribution 4.0 International License.
SecDev '26, Montreal, QC, Canada
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2602-6/2026/07
https://doi.org/10.1145/3805773.3805996

1 Introduction

Fifty years ago, symbolic execution (SE) was created as an automatic comprehensive program testing technique to simulate the execution of a program using symbolic inputs, verify the program behavior with the symbolic constraints, and find potential inputs triggering violations by solving the symbolic constraints [124]. In the decades since, it has evolved significantly and become one of the most foundational techniques in the domain of system and software security. It is considered "highly effective" [39, 190] for program testing and is widely applied to a variety of security applications, including vulnerability discovery [181, 195, 225, 281], exploit generation [19, 20, 47, 97, 201], shellcode transplantation [26], function similarity analysis [147, 157], crash triaging [269, 270], and automated patching [216, 259].

As we mark half a century of SE, this paper presents a new perspective to the research: not simply to catalog or synthesize SE's past, but to chart a course for its *future*. While prior surveys and systematizations have mapped the evolution and diversification of SE techniques [24, 218], we ask a different research question: *How to design SE to foster the current and future research in and with SE?* We consider two research goals:

- RG1 Enable the continued advancement of SE for itself and its existing applications.
- RG2 Facilitate the expansion of SE's application into new problem domains and analysis techniques.

To tackle this question, our insight is that, as existing high-quality literature already frame their specific SE-related research problems well, we can leverage these well-defined problem formulations to decompose the SE framework into modules. Such a modular design naturally accommodates integration of prior work and enables future research to more easily compare against existing approaches by plugging in and experimenting with previously integrated components. Based on this insight, we systematically review the SE-related research published at the top-tier proceedings in cybersecurity and software engineering. We collected 225 relevant high-quality papers from the past 15 years that advance, diversify, or adopt SE techniques. We perform a qualitative analysis to identify themes in the literature and distill recurring design patterns from which we synthesize five common SE modules that

existing research: *Execution Driver*, *Search Heuristic*, *Symbolic State Manager*, *Environment and Memory Model*, and *Constraint Solver*. We present the techniques related to each module, as well as use cases. We also show trends in the literature related to each module and highlighting areas for improvement (RG1).

For RG2, as future program domains and analysis techniques are inherently unknown, we instead propose to focus on *SE tool usability improvement*. That is, what research is needed to improve each module to allow future security practitioners to more easily apply SE to emerging problems. To this end, we first perform a concurrent systemization of literature on human factors research focused on software analysis tool users, i.e., developers and security professionals (DSP). We consider research which investigates how these users understand code and use current tooling for program analysis. Through the qualitative review of 66 DSP-focused papers from HCI, cybersecurity, and software engineering venues over the past 15 years, we enumerate common guidelines for usable tool development to meet DSP users' information and interaction needs. We then compare these guidelines against the SE literature using our modular framework to identify usability gaps and highlight directions for future work.

Contributions. Our key contributions are:

- A thematic analysis of 225 papers on SE, which identified five SE modules, current approaches to each and future work opportunities.
- A second thematic analysis of 66 papers on human factors affecting DSP users, which enumerated six guidelines for software analysis tool usability.
- An outline of how our SE modules guide future work to meet each usability guideline, demonstrating the module framework's utility.

2 Background & Related Work

Symbolic Execution (SE). SE executes a program with *symbolic inputs* instead of concrete values. This enables reasoning about multiple feasible execution paths simultaneously. Starting from the program's entry point, whenever SE encounters a conditional branch whose outcome depends on the input, the SE engine queries a Satisfiability Modulo Theories (SMT) solver with the collected path constraints for both directions to determine which paths are feasible. If both directions are feasible, the engine forks the execution to explore each path separately. When an execution path reaches an exit point, the symbolic execution engine queries the SMT solver for a solution to the full set of path constraints, then generates a concrete input that follows that specific path. This process is repeated for each path explored. Formally, the SE engine maintains the following Symbolic State Information (SSI):

σ: The *symbolic state*, a mapping from program variables to their symbolic expression.

π: The *path constraint*, a quantifier-free first-order formula over symbolic expressions.

Limitations. Traditional SE allows for the exploration and reasoning of multiple execution paths simultaneously. However, this has proven costly and problematic when trying to analyze large, complex targets. The four primary issues are:

1) Path Explosion. SE forks execution at every feasible branch point to explore possible paths concurrently. Loops and (recursive) function calls [24, 40] are prevalent in real-world programs, significantly increasing branch points. As the number of paths grows exponentially, SE quickly exhausts computational resources. This is commonly referred to as the *path explosion* problem.

2) Costly Constraint Solving. Prior works [54, 189] have identified constraint solving as a key performance bottleneck in SE. Constraint solving is inherently expensive, and the solving rate drops as the exploration deepens and path conditions become more complex [260]. Combined with path explosion, this amplifies the number of solver queries and computation overhead.

3) Path Divergence. Despite solving path constraints, the generated input can diverge from the desired path due to incorrect symbol propagation or engine-internal imprecision, e.g., when handling floating point numbers [24, 57].

4) External Function and Environment Modeling. Real-world programs frequently invoke external libraries or system calls that are difficult for SE to model precisely. Lacking access to their internal logic, SE relies on approximations, leading to inaccuracies in path constraints, undermining SE's effectiveness.

Dynamic Symbolic Execution. One prominent approach to mitigating the above limitations is Dynamic Symbolic Execution (DSE), also known as concolic execution [1]. Given a target program, DSE executes it with a concrete input while symbolically tracing the path. At each symbolic branch point, it collects the path constraints of the untaken branch and, if satisfiable, generates a new input to explore that path. This process is repeated for each new input.

DSE overcomes some of SE's shortcomings: By exploring one path at a time, it delays path explosion and avoids immediate resource exhaustion, though the exponential path space remains a long-term challenge manifesting as the number of generated inputs grows. It simplifies constraint solving through partial concretization, resulting in smaller and more tractable path constraints. However, the iterative generation of new inputs still requires repeated solver calls, so the overall cost remains high for deep or complex paths. DSE benefits from concrete execution by using real values for external calls and environment interactions, reducing the need for symbolic approximations. Nonetheless, if such calls produce nondeterministic behavior or symbolic dependencies, DSE may still struggle to maintain path completeness.

Related Work. Starting in 1976 [124], SE has developed into an indispensable software testing technique. Given its crucial role, prior work has reviewed the field. In 2009, Păsăreanu and Visser [179] conducted an early survey. One year later, Schwartz et al. [210] provided a seminal explanation of SE. One year after, Cadar et al. [39] briefly summarized the state of symbolic execution, including its use in practice. Cadar and Sen [40] then followed up on this work in 2013, reviewing the progress of SE across the past three decades. In 2018, Baldoni et al. [24] surveyed the still advancing field, followed by the systematic comparison of intermediate representations used by different SE engines by Poeplau and Francillon [188] in 2019. The same year, Borzacchiello et al. [30] systematized how SE engines model memory as part of their work. More recently, Bailey and Nicholas [23] surveyed how SE can be used for malware, firmware,

[1] While some works distinguish the two, most use the terms interchangeably.

or protocol analysis. In contrast, we seek to bridge the gap between SE tools and their users, an understudied aspect so far. To this end, we not only systematically capture the improvements to core elements of symbolic execution, but we focus on the usability, a key concern impeding broader adoption.

The past decade has also seen extensive research on the *human factor*. Tahaei and Vaniea [229] surveyed developer-centric security, studying how developers write security-relevant code and tools they use. In 2021, Kaur et al. [119] reviewed ten years of research on human factors and Mokhberi and Beznosov [159] systematized the human, organizational, and technological challenges impacting developers while attempting to write secure code. One year later, Rajapakse et al. [192] systematically reviewed challenges preventing DevSecOps adoption and in 2023, Nurgalieva et al. [172] reviewed and categorized factors influencing developers' security considerations. These works focus on the developers and their challenges in writing secure code, while we focus on SE as technique and produce a combined list of *usability guidelines* to enable widespread adoption of SE as testing technique.

3 Systematization Methodology

In this section, we describe our methodology, including how we identified SE and DSP papers, qualitatively coded papers to identify themes, and our analysis of relationships between SE and DSP work to guide future module-specific usability improvements.

3.1 Paper Selection Process

SE paper selection. We began systematizing existing research on SE by collecting publications which described SE from major academic cybersecurity and software engineering venues. We considered IEEE Security and Privacy (S&P), USENIX Security, ACM Conference on Computer and Communications Security (CCS), Network and Distributed System Security Symposium (NDSS), ACM Foundations of Software Engineering (FSE), International Conference on Software Engineering (ICSE), and ACM International Symposium on Software Testing and Analysis (ISSTA). For each venue, we searched the past 15 years for relevant SE papers. To assess relevancy, we searched each paper's text for all possible combinations of *"symbolic"*, *"dynamic symbolic"* and *"concolic"* with *"execution"*, *"verification"*, and *"testing"*. For example, this included *"dynamic symbolic verification"*, *"symbolic execution"*, and *"concolic testing"*. These terms were selected by research team members with over a decade of SE research experience and multiple publications on SE.

Then, we searched Google Scholar and performed a Google search using the same keyword combinations. We added publications listed in a GitHub page created to highlight important work in SE [265][2]. As we reviewed each result, we looked for common terms specific to SE that could be used to broaden our search but did not observe any unique terms beyond those included initially.

With our initial SE corpus, we reviewed each paper's citations, adding cited papers about SE for subsequent review. This process was repeated for every paper until we found no new papers. We collected 287 papers and repeatedly encountered duplicate citations already in our corpus. We restrict the papers about SE to the last 15 years to capture changes in focus from the research community.

[2]This list include 102 SE papers

DSP paper selection. We followed a similar process for the DSP corpus, focusing on papers that described how various users (e.g., developers and malware and vulnerability analysts) analyze programs and use program analysis tools. We first collected publications from the same set of cybersecurity and software engineering venues from the SE search. We expanded our search to include human-computer interaction (HCI) venues, which are more likely to publish human-centric studies, including the Symposium on Usable Privacy and Security (SOUPS), the Conference on Human Factors in Computing Systems (CHI), the International Symposium on Empirical Software Engineering and Management (ESEM), the Special Interest Group for Computer Science Education Conference (SIGCSE), the PLATEAU Workshop, and the ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH). For each venue, we performed a keyword search of each paper for *"interview"*, *"participatory design"*, *"usability"*, *"human and computer interaction"*, and *"qualitative"* to identify papers investigating human subjects. We manually reviewed each to determine whether it studied DSP users. After identifying an initial batch of developer publications, we reviewed each paper's citations, adding any discussing DSP users' processes. Ultimately, this resulted in an 84 paper corpus.

3.2 Paper Exclusion Criteria

SE exclusion criteria. To refine our SE corpus's focus, we only considered papers where SE has been used in practice and implemented technical improvements to address SE's common limitations, thus demonstrating directly leveragable advancements to SE. We removed papers that did not directly implement their contributions in a SE tool or use SE as a component of a larger system. This excludes papers using symbolic reasoning or ideas from SE. For example, Ferles et al. used symbolic reasoning to automatically insert explicit signals into concurrent programs to prevent concurrency errors [81]. While they implement their ideas in a tool called Expresso, they only adopt the idea of symbolic variables but do not symbolically execute the program, instead using other static techniques. After applying our exclusion criteria, 225 papers remained.

DSP exclusion criteria. For our DSP corpus, we investigated how people perform program analysis and how software analysis tools can be designed to fit users. We only considered papers investigating program analysis users' processes, including interview studies, observations, formative design studies and controlled experiments. To capture a range of program analysis users, we reviewed literature about developers, professionals specializing in vulnerability discovery (e.g., red teams, bug bounty hunters), and malware analysts. We did not include papers on tasks these users performed beyond their interactions with code, such as Alomar et al.'s interviews with managerial security professionals, which studied non-technical challenges managing bug bounty programs [11]. After pruning these types of papers, we had 66 papers.

3.3 Qualitative Corpora Analysis

Each set of publications, i.e., SE and DSP corpuses, were analyzed separately, but we performed a similar iterative, open coding [226,

pg. 101-122] for each. For the SE codebook, we followed an inductive approach where two researchers jointly reviewed 10 SE papers to develop and initial codebook, allowing themes to emerge from the data. The results of this phase were then discussed amongst the research team. One of the two researchers performing the coding is an expert in program analysis with experience working with SE and the full research team includes multiple experts in SE with several related papers published. We sought to capture what elements of SE best fit each paper's contributions such that just our applied codes can describe the contributions at a high-level. Therefore, we labeled the SE module each paper improved, which SE limitations it improved upon, how the authors evaluated the paper's contribution, what specific use cases were considered, and any additional program analysis techniques the work combined SE. The two researchers then independently coded publications in rounds of 20. After each round, the researchers compared codes, resolved disagreements, and updated the codebook as needed. Any codebook changes were applied to previously coded papers. We also calculated Krippendorff's alpha (α) to assess codebook inter-rater reliability (IRR) as it is a conservative measure, accounting for chance agreements [101]. We did not calculate α for objective codes like the year the paper was published or the base SE engine (e.g., angr, KLEE) used, as these can be taken directly from the text [153]. After five rounds of independent coding, α exceeded 0.8 for each code, indicating strong agreement [101]. The remaining 115 papers were divided evenly between the two researchers and coded by a single coder. The final SE corpus codebook can be found in Appendix A.

Next, we conducted an inductive qualitative coding of the DSP corpus. For this corpus, we had a more narrow focus, only seeking to identify prior design guidelines for DSP users' program analysis tools. Two researchers jointly reviewed five DSP papers to create the initial codebook. The results of this phase were then discussed amongst the research team. One of the two researchers performing the coding is an expert in usable RE tool analysis and the full research team includes multiple experts in human factors research with multiple relevant publications focused on software analysis usability. The two researchers then independently coded papers in rounds of 10, meeting after each round to discuss disagreements, reach consensus and update the codebook. We reached a sufficient agreement ($\alpha = 0.87$) after four rounds. The remaining 21 papers were coded by a single coder. Table 6 shows the final codebook.

After completing open coding, we performed axial coding to identify relationships within and between codes to produce higher-level themes [226, pg. 123-142]. In particular, we identified common characteristics of papers focused on each SE module in the SE corpus and produced a set of overarching guidelines for usable software analysis tool development to support DSP users from the DSP corpus. Finally, we investigated connections between codes from the two corpora to understand how existing SE work could be advanced to meet the identified usability guidelines.

3.4 Limitations

Though we performed a thorough review to produce our corpora, we may have missed papers from less know venues not cited by a paper in our corpora. Because of our thorough search, we expect the number of missed papers is minimal. Also, because we reached thematic saturation, i.e, we stopped seeing new themes [49, pg. 113-115], we expect we enumerated all relevant themes.

Additionally, we chose only to include SE papers with implementations. This leaves out publications whose main contributions are proofs of correctness or purely theoretical, building on concepts from SE. Including these papers could better describe SE research's current state of the art. However, because our focus is on improving practical tool development for SE, the lack of implementation limits the insights we can draw. Instead, our focus on grounded implementations paired with our systematization of DSP tasks delivers rich insights and future directions for SE research.

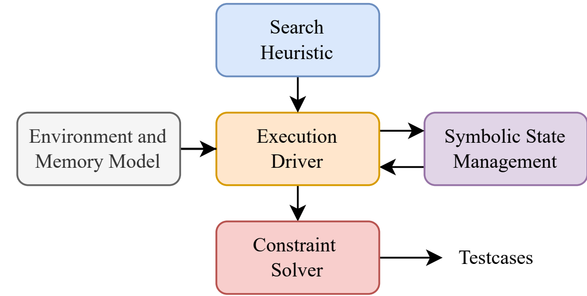

Figure 1: Proposing symbolic execution modules

4 Modularizing SE by Existing Literature

Based on the qualitative study on the existing literature that improves or adopts symbolic execution, we formally categorize SE into five widely recognized modules as shown in Figure 1. In this section, we will present each module and the associated work, and we provide suggestions on how a research-friendly SE framework should be designed. For clarity, we refer to an existing running example [24] to illustrate prior literature.

4.1 Running Example

The running example is shown in Listing 1. Here, a developer might want to check whether the *assert* statement on line 8 could fail. The function `foobar` takes two integer values, each potentially storing 2^{32} distinct values.

```
1 void foobar(int a, int b) {
2     int x = 1, y = 0;
3     if (a != 0) {
4         y = 3 + x;
5         if (b == 0)
6             x = 2 * (a + b);
7     }
8     assert(x - y != 0);
9 }
```

Listing 1: Running example by Baldoni et al. [24].

SE now executes our program with *symbolic* inputs, i.e., the symbolic state $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b\}$. Here, α_a and α_b represent the entire space of 32-bit integers. The initial path constraint $\pi = true$. Line 3 then contains a branching point with two outgoing branches. SE queries the solver with both branches' path conditions ($\alpha_a \neq 0$ for the *true* and $\alpha_a == 0$ for the *false*) to assess feasibility. Both are feasible, so SE forks the execution to pursue both directions. Line 5

	Category	#	References
a.	Selective Invocation	20	[42, 50, 53, 58, 62, 67, 91, 109, 112, 146, 161, 217, 236, 237, 242, 256, 262, 275, 287, 293]
b.	Forking Policy	15	[36, 62, 94, 106, 116, 143, 160, 215, 241, 247, 266, 272, 276, 282, 284]
c.	Instrumentation	7	[54, 70, 87, 122, 189, 211, 228]

Table 1: Execution driver paperss

introduces another branching point with both branches being feasible. After this point, SE pursues three paths concurrently. When considering the path where both branches evaluate to *true*, i.e., π evaluates to $(\alpha_a \neq 0) \wedge (\alpha_b == 0)$, SE adds $x \mapsto 2 * (\alpha_a + \alpha_b)$ to the symbolic state σ at line 6. Only on this path, the outcome of the `assert` in line 8 depends on symbolic values. SE then attempts to verify the *assert* on line 8 by asking a constraint solver if its condition always holds given the current x and y values. The solver reports the condition does not always hold and provides a counterexample, e.g., $\{a = 2, b = 0\}$. The analyst can conclude there exist values of a and b that trigger the assertion.

4.2 Execution Driver

Execution Driver is the central driving component of Symbolic Execution. Returning to our example in Listing 1, the *Execution Driver* emulates each instruction while maintaining the symbolic memory state and path condition. At each conditional branching point, it explores both feasible directions by forking the execution, thereby pursuing all viable paths concurrently. This traditional approach works well for our running example which has only two branching points and no loops. It does not scale well to real-world programs that typically contain many functions and loops, causing the number of execution paths to grow drastically. With finite computational resources, this growth quickly becomes unmanageable, resulting in the *path explosion* issue. We classify and summarize publications that improve the *Execution Driver* module in Table 1.

Selective invocation. Twenty papers reduced SE overhead by *selectively* invoking the *Execution Driver* on a *program subset* rather than the full program, as summarized in Table 1.a. For example, Xiao et al. [262] use input values and functions referenced in vulnerability reports to seed and apply selective concolic execution.

Forking policy improvements. Fifteen papers boosted the efficiency of evaluating multiple different program states simultaneously (Table 1.b). Liu et al. [143] reuse overlapping SSI of different paths to concurrently run the *Execution Driver* in the same environment, thus reducing memory costs s exploring multiple states; this includes integrating a symbolic and concrete execution engine under the same OS process with a shared environment. To ground this with our example programs in Figure 1, the path constraints of the two new execution paths created at line 6 of *foobar* share a common clause: $\alpha_a \neq 0$, added at line 3. Liu et al. propose sharing this information as a single memory object between concurrent SE threads and concrete execution engines to reduce memory overhead and detect divergent paths.

Reduced dynamic instrumentation. Seven papers instrumented programs before or at compilation time, or using binary translation (Table 1.c) to avoid costly dynamic binary instrumentation. SymQEMU [190] hooks into QEMU's binary translation to emit symbolic-handling logic directly as machine code, such as inserting guiding constraints or skipping uninteresting code (e.g., standard library functions). The authors propose SYMCC [189], a compiler that embeds concolic execution logic into binaries at compile time. Coppa et al. [70] take this one step further and propose SYMFUSION, a concolic executor mixing static, compile-time instrumentation and dynamic instrumentation at execution time.

	Category	#	References
a.	Path Pruning	40	[2, 7, 12, 22, 37, 38, 42, 46, 52, 59, 62, 63, 66, 92, 107, 109, 116, 121, 128, 143, 148, 154, 162, 175, 178, 182, 184, 200, 207, 211, 247, 249, 262, 267, 272, 276, 277, 279, 283, 290]
b.	Path Selection	36	[6, 10, 12, 25, 33, 38, 42, 43, 45, 47, 51, 60, 66, 78, 102, 106, 136, 137, 142, 148, 149, 151, 167, 176, 184, 191, 193, 199, 217, 223, 227, 237, 268, 280, 285, 289]
c.	Path Consolidation	9	[34, 109, 162, 212, 214, 235, 241, 250, 282]
i.	Prioritization Metrics	29	[7, 22, 25, 33, 34, 42, 43, 45, 52, 56, 59, 60, 109, 142, 149, 178, 182, 184, 247, 249, 250, 262, 267, 268, 277, 283, 285, 289, 290]
ii.	Path Constraints	22	[43, 45–48, 51, 52, 62, 63, 109, 162, 163, 175, 178, 200, 207, 212, 272, 276, 277, 279, 290]
iii.	Points of Interest	12	[52, 66, 115, 146, 151, 167, 191, 193, 199, 217, 280, 285]

Table 2: Search heuristic papers, outlining both techniques (a.–c.) and metrics guiding the search (i.–iii.)

4.3 Search Heuristic

Available resources in terms of CPUs usable for concurrent path exploration are usually quickly exhausted during SE due to frequent forking. Thus, SE needs a *search heuristic* to prioritize interesting paths to explore. For instance, assume we have a single CPU core and encounter the branching point in line 3 in our running example in Listing 1. Then, the search heuristic would decide branch prioritization. Table 2 shows publication statistics.

Path prioritization via pruning / selection / consolidation. Eighty-five papers use path prioritization to improve the *Search Heuristic* (Table 2.a-c). Forty prune irrelevant execution paths from the search space, 36 made advancements in path selection policies, and 9 used a mix of path pruning and prioritization. These prioritization approaches use a range of metrics to make decisions.

Prioritization metrics. Twenty-nine papers use program information to inform the *Search Heuristic* (Table 2.i). Cha and Oh [45] score paths on 40 features, including the presence of pointer expressions, function calls, and nested branches, to refine their *Search Heuristic*. Permenev et al. [182] utilize SE for testing Ethereum smart contracts, which rely on a metric called *gas* that measures the computational effort of executing operations. They prune potential paths by symbolically tracking gas costs and terminating infeasible execution paths triggering out-of-gas exceptions.

Path Constraints. Twenty-two papers guide path exploration by evaluating path constraints' feasibility and solving cost (Table 2.ii). For example, Muller et al. [162] check for inconsistent π constraints that would result in unreachable program states. Similarly, Luo et al. [148] use machine learning to estimate path constraint solving cost and direct SE towards the path with the lowest cost.

	Category	#	References
a.	In-memory Storing	25	[32–34, 54, 63, 69, 72, 83, 84, 98, 117, 137, 161, 162, 171, 176, 191, 198, 200, 211, 235, 238, 250, 267, 292]
b.	Memory Allocation	11	[36, 38, 42, 47, 69, 83, 144, 189, 206, 237, 238]

Table 3: Symbolic state manager papers

	Category	#	References
a.	Modeling improvements	21	[8, 31, 34, 35, 59, 67, 71, 76, 96, 112, 130, 131, 134, 143, 145, 146, 182, 199, 248, 261, 275]
b.	Function stubs	18	[2, 36, 50, 59, 75, 91, 123, 130, 133, 134, 139, 154, 156, 166, 242, 243, 273, 286]

Table 4: Environment and memory model papers

Known points of interest. Twelve papers selectively use SE to explore a particular point in a larger program (Table 2.iii). These points of interest are known *a priori*. Johnson et al. [115] utilize a context-sensitive distance function to calculate the distance from the current execution path to a security-critical point deeper inside firmware binaries. Similarly, Yu et al. [280] leverage static and dynamic analysis to evaluate future branches against previously explored ones to calculate the probability that a new path will satisfy a predefined property of the program.

4.4 Symbolic State Manager

An orthogonal approach to mitigating *path explosion* is to refine the *symbolic state manager* to use less resources during execution. Although this does not address the problem, it helps mitigate or delay its effects, enabling the analysis of more complex programs. We capture publication statistics in Table 3.

Symbolic state in-memory representation. The most common *Symbolic State Manager* improvement (25 papers) optimized aspects of SE by using creative data structures to store SSI (Table 3.a). Cho et al. [63] utilized instruction-level taint analysis to build field transition trees for each inferred field used in symbolic execution. This tree representation can solve simple constraints, thereby only selectively invoking the *Constraint Solver*, which can be another bottleneck of SE as we will discuss further in Section 4.6. Similarly, Coppa et al. [69] used a memory-wise paged interval tree to share SSI between different execution states, minimizing duplicate information between partially overlapping execution paths.

Symbolic states memory allocation. Across our corpus, there were eleven papers that adjusted how symbolic states are managed in program memory (Table 3.b). Busse et al. [38] save symbolic states to disk, which can be restored later to run SE indefinitely. Schemmel et al. [206] propose a deterministic memory allocator for dynamic symbolic execution, where each execution path individually allocates memory instead of relying on a global allocator for all execution paths. This allows for cross-path determinism and increased stability across multiple execution instances.

4.5 Environment and Memory Model

Whenever the *Execution Driver* updates the SSI for an execution path dependent on factors outside the execution context's scope, e.g., environment variables or network connections, it uses the information in the *Environment and Memory Model* to dictate how it should progress. If line 4 in Listing 1 of *foobar* set y to be a value received over a previously established network socket: `y = socket.recv(4)`, the received value can be any 32-bit integer. However, if we know only values from 0 to 100 are sent, we can symbolically define this expected behavior in the *Environment and Memory Model*. These behaviors are not easily defined in practice, especially when analyzing malware or programs with memory-intensive operations, whose size and addresses are symbolic.

Memory value modeling improvements. Twenty-one papers made advancements in modeling memory values that can generalize to various SE applications (Table 4.a). *MINT* defers any potentially expensive symbolic reasoning about memory accesses by tracking a memory state timeline thorughout execution [31]. Once *MINT* encounters a high memory load, it restores previous memory states to reason about potential valid memory operations. Chen et al. employ a more common technique, creating a memory model of expected program behaviors based on concrete memory address ranges from prior execution traces from a concolic fuzzer [59].

Function stubs. To simplify *Environment and Memory Model* considerations, 18 papers created stub functions which are simplified versions modeling function behavior without needing to be symbolically executed (Table 4.b). Vishnyakov et al. construct symbolic formulas for standard library functions' return values [242]. Similarly, Nasrabadi et al. abstracted cryptographic libraries, attacker calls, and random number generators [166].

	Category	#	References
a.	Simplification	30	[2, 18, 34, 44, 48, 50, 51, 55, 56, 61, 62, 87, 93, 94, 108, 113, 115, 135, 156, 166, 167, 177, 189, 211, 212, 223, 241, 249, 252, 276]
b.	Representation	16	[7, 47, 51, 56, 63, 64, 95, 107, 182, 211, 215, 223, 253, 274, 282, 284]
c.	Caching	18	[2, 15, 38, 63, 72, 95, 107, 111, 115, 116, 184, 189, 205, 219, 249, 272, 284, 290]

Table 5: Constraint solver papers

4.6 Constraint Solver

Finally, the *Constraint Solver* is invoked whenever the *Execution Driver* needs to resolve symbolic conditions. When the *Execution Driver* encounters the conditional branches on lines 3 and 5, it invokes the *Constraint Solver* to check if each path condition has valid inputs that make the condition true or false. This is typically accomplished by a Satisfiability Modulo Theories (SMT) solver. Here, the *Constraint Solver* is also invoked at the *assert* statement on line 8, which is both our point of interest and a critical instruction that can cause the program to halt. Invoking the *Constraint Solver* is generally a main SE bottleneck. Solving queries is costly compared to other steps, and, the path explosion problem makes these constraints difficult to solve for more complex programs.

Constraint simplification. Thirty papers concretized symbolic values, pruning redundant constraints, or rewriting constraints to reduce *Constraint Solver* query complexity (Table 5.a). Cha et al. selectively symbolized variables and concretizing others to reduce the search space and eventual constraint complexity [44]. A shortcomings of concretizing values is a code coverage loss and path divergence. To combat this, Pandey et al. deferred concretization by preserving concrete and symbolic values for all variables [177].

Constraint representation. Sixteen papers optimized constraint representations to reduce constraint solving cost (Table 5.b). Wen et al. defined a domain-specific language to build *Token Flow Graphs* to describe potential smart contract behavior, allowing more efficient symbolic execution when searching for logical errors [252]. Alternativley, Choi et al. approximated path constraints with valid intervals instead of the full range of values [64].

Constraint solution caching. Eighteen papers leveraged solved constraints to sidestep the *Constraint Solver* (Table 5.c). Kapus et al. prioritized exploring execution paths whose conditions were deemed satisfiable previously by the *Constraint Solver* [116]. Checking that an assignment satisfies a formula is less expensive than generating a fresh assignment, which requires a *Constraint Solver* query, and thus their approach reduces constraint-solving cost.

5 Systematizing Developer & Security Professionals' Tasks and Information Needs

Now, we turn our attention to considering the challenges and proposed improvements to security tooling from prior literature and its application to SE. We organize this section around six guidelines for usable software analysis tool development drawn from the literature. For each, we give examples of the developer information and interaction needs found in prior literature. We conclude by applying the guidelines to our SE modules.

Integrate with workflow (G1). A common theme in the DSP papers was the importance of fitting in with the users' current code review approaches (N=56). This is an expected usability guideline for any tool, but is especially true for DSP tasks, which require specialized knowledge and are typically completed over a long period of time. Further, our review identifies several workflow aspects that must be considered for tool usability.

First, tool designers should ensure program analysis tools fit DSP users' processes (N=15). Votipka et al. found software reverse engineers' (REs) processes could be divided into three phases and recommended tools bridge these phases to limit users' cognitive burden during phase transitions [245]. Unfortunately, there does not appear to be a single process for all DSP user groups. While Votipka et al.'s proposed process model fits similar RE work, such as hardware reverse engineering [28], Katcher et al. found it required some modification when studying protocol REs [118]. Similarly, in Ardi et al.'s investigation of threat modeling integration throughout the software development lifecycle found different tools worked better for different phases (e.g., initial development or threat model maintenance). Therefore, they recommended specific tools' use cases should be clearly defined and introduced to users at the point in their process where they fit best [16].

Software analysis tools should also selectively present information (e.g., results, suggestions) to avoid interrupting users (N=23). In interviews with developers deciding whether to use analysis tools, Johnson et al. found a key deterrent was whether the tool "disrupts your flow," with participants recommending tools wait for idle times to present outputs [114]. Piskachev et al. recommended tools allow users to selectively enable rules for static vulnerability detection [185]. This was also evidenced in Sadowski et al.'s deployment of the Tricorder static analysis system, which showed the best time to provide analysis results was during code review, when developers were already accustomed to receiving feedback [203].

Finally, to best fit DSP users' workflows, tool developers need to integrate with existing tooling (N=37). For security analysts, this means integrating with packet capture tools, decompilers, fuzzers, and other components of users' analysis toolchains. For developers, software analysis tools should integrate with the compilers (N=6) and IDEs (N=25) developers use. Christakis et al. found in developer interviews and surveys that there was a clear preference for security analysis results to be presented in the IDE [65]. Conversely, other work showed integrating with the compiler may be preferable, as IDEs are not universally used [114, 203].

Easy to get started, but ability to go deep (G2). Another common theme was that software analysis tools should adapt as user learns (N=40). That is, tools should make it easy for beginners to get started (N=19). In Plöger et al.'s usability study of CS students using libFuzzer and the Clang Static Analyzer, students could not get libFuzzer working, limiting its utility [186]. Plöger et al. then compared libFuzzer and AFL's usability to similar results [187]. These papers suggest using GUIs or other aids to guide users through common tasks and limit their options as beginners learn.

However, prior is clear that DSPs would not find a tool useful if it only provided common functions. As users gain expertise, tools should allow users to customize their experience (N=28). In Wong et al.'s study of professional malware analysts' workflows, they observed "*lack of customization of tools is the main disadvantage that participants mentioned in the interviews*" [278]. Addressing this often requires providing an API or command-line interface with greater expressiveness than typically possible through a GUI.

Informative results are critical (G3). In the next set of guidelines, we outline *how* analysis tools should provide information. The most important of these guidelines, and the most common design suggestion overall, is that a tool's results should be easy for users to process and rely on (N=70). This may seem obvious at first glance. A prior review of reverse engineering (RE) tools by Mattei et al. found the plurality of existing RE tools focus primarily on improving code readability [152], as this is a central challenge in reverse engineering [246]. However, prior literature also shows providing truly informative results can be challenging.

To meet this guideline, prior work recommended providing examples and explanations to demonstrate final output reasoning (N=25). Interviews with security analysts developing and deploying machine learning tools for network security analysis by Mink et al. revealed these users wanted tools to provide explanations, so they could have confidence in the results [158]. Similarly, in a survey of developers, Do et al. found participants wanted vulnerability detection tools to give exploitation examples for flagged vulnerabilities so they could validate the exploit feasibility [77].

Several developer-focused papers have suggested analysis tools provide fixes on bug identification (N=13). Johnson et al. found developers were less likely to use a bug detection tool if it did not fix suggestions [114]. Bug fixes provide additional context, helping users better understand the tool's result to determine whether they should trust the result and quickly take appropriate actions.

Along with providing enough information for user verification, tools should ensure accuracy to minimize this verification burden.

Category	#	References
Integrate into workflow (G1)	56	
Fit users' process	15	[9, 16, 21, 27, 28, 65, 114, 118, 140, 180, 185, 186, 220, 245, 257]
Avoid interrupt user workflow	23	[17, 21, 27, 65, 79, 90, 110, 114, 127, 140, 141, 150, 169, 183, 202, 203, 224, 233, 234, 245, 255, 258]
Integrate with other analysis tools	37	[5, 9, 14, 16, 21, 29, 65, 68, 77, 85, 86, 99, 104, 105, 110, 114, 118, 118, 141, 158, 158, 165, 169, 185, 196, 203, 208, 213, 221, 231–234, 245, 255, 258, 263, 288, 291]
Integrate with compiler	6	[68, 73, 110, 114, 203, 213]
Integrate with IDE	25	[5, 9, 14, 16, 21, 29, 65, 68, 77, 80, 85, 86, 104, 105, 110, 114, 118, 158, 165, 169, 185, 196, 203, 208, 213, 221, 231–234, 245, 255, 258, 263, 291]
Easy to get started, but ability to go deep (G2)	40	
Make it easy to start using tool	19	[9, 14, 29, 41, 82, 86, 105, 110, 120, 127, 140, 164, 168–170, 186, 232, 255, 288]
Support customizability	28	[3, 21, 27, 65, 73, 74, 77, 79, 80, 110, 114, 140, 165, 169, 170, 174, 186, 203, 221, 222, 224, 232, 233, 240, 257, 263, 278, 288]
Informative results are critical (G3)	70	
Provide examples to demonstrate reasoning	25	[3, 17, 68, 77, 88, 89, 114, 126, 155, 168–170, 174, 180, 185, 186, 194, 197, 208, 209, 221, 230, 233, 244, 291]
Provide bug fixes	13	[16, 65, 77, 114, 169, 202, 203, 213, 221, 224, 233, 263, 291]
Avoid alert fatigue	14	[5, 14, 21, 27, 73, 105, 114, 164, 185, 196, 202, 204, 220]
Minimize false positives	26	[5, 13, 14, 16, 21, 65, 77, 100, 103, 110, 114, 132, 140, 186, 196, 203, 204, 221, 224, 230, 232, 233, 254, 255, 257, 263]
Support output verification	30	[9, 14, 16, 27, 65, 79, 80, 82, 88, 90, 100, 110, 114, 140, 168–170, 196, 202–204, 213, 221, 231–233, 240, 255]
Improve documentation	30	[3, 4, 27, 68, 74, 82, 86, 105, 120, 126, 127, 138, 155, 164, 168, 170, 180, 186, 194, 196, 197, 208, 221, 233, 244, 251, 258]
Interactive use; with human in tight loop (G4)	28	
Preview changes before implementing	5	[41, 65, 114, 169, 221]
Analysis should avoid slowing down the user	22	[5, 65, 77, 79, 90, 114, 120, 125, 165, 169, 170, 180, 185, 203, 204, 213, 224, 240, 255, 263, 264, 291]
Provide input and output in the context of code (G5)	17	[21, 65, 85, 114, 118, 129, 158, 174, 221, 224, 232–234, 245, 263, 288, 291]
Support collaboration (G6)	23	[16, 17, 21, 65, 77, 90, 99, 114, 173, 183, 186, 203, 204, 208, 221, 222, 231, 239, 240, 251, 255, 258, 264]

Table 6: DSP guidelines. We present the six main guidelines in bold at the beginning of each section of the table. For guidelines with identified sub-guidelines, for brevity, we do not list all the papers associated with that guideline on the top row. The associated papers for those guidelines are the union of each paper set associated with the guidelines sub-guidelines. Note, the counts do not sum to the total number of papers as each paper could discuss multiple guidelines.

The literature is clear false positives are particularly problematic as they lead to alert fatigue (N=14) and cause developers to cease tool use (N=26). In Sadowski et al.'s deployment of the Tricorder, they found developers would quickly abandon high false positive static analysis tools [203]. While false negatives are also problematic, prior work suggests DSP users prefer to perform their complex tasks manually to fill in the analysis' gaps rather than spend time assessing analysis result validity. Since any sufficiently complex analysis will produce errors, many papers recommended providing tools and information to help DSP users validate tool output (N=30). For example, when reviewing GitHub issues submitted by developers using OSS-Fuzz, Nourry et al. observed developers found it difficult to reproduce OSS-Fuzz-identified crashes, needing more information about the crashing execution trace [170].

Finally, many papers described needing improved documentation (N=30). If users cannot interpret analysis results out of the box, they must be able to find some documentation explaining the results to validate tool findings or debug issues. Barke et al. found developers regularly reference API documentation to validate Github Copilot AI-generated code [27]. However, they found Copilot itself did not provide sufficient documentation to clarify its operation, leading to misconceptions about how Copilot makes suggestions.

Interactive use; with human in tight loop (G4). Software analysis tool interactions should be designed to allow DSP users to review the tool's results, then easily reconfigure and re-run the analysis (N=28). While evaluating ReCode, an automatic code transformation tool, Ni et al. found developers preferred incremental transformations [169]. In interviews with protocol reverse engineers, Katcher et al. found they preferred tools make smaller suggestions (e.g., labelling possible datatypes instead of automatically inferring the full protocol) [118]. This was because these participants did not trust tools to perform the complete task and preferred a tight iterative loop with the tool so they could understand what it was doing and be able to identify issues and intervene manually. Another way to support this interactive use is for tools to preview code changes or allow users to quickly undo tool actions (N=5).

It is also important that analyses be conducted quickly (N=22). As several iterations are often needed, a slow analysis will introduce significant overhead and prior work suggests they will abandon the tool. Several prior studies found tool speed was a primary factor in users' decisions not to use a software analysis tool [65, 114, 185, 204]. Therefore, sacrificing some specificity or scope of analysis for speed seems appropriate to ensure continued use.

Note, this guideline has an interesting interaction with G1, which suggests analysis tools fit users' workflow and work across task phases. Minimizing the scale of tasks an analysis performs suggests each process phase be completed separately, i.e., its minimal components. However, context switching between the user and tool can sometimes be jarring. Votipka et al. found it is challenging for reverse engineers to switch between static and dynamic contexts who must manually transfer information between the two contexts [245]. They recommended tools bridge this barrier. Our literature review also suggests analysis tools should bridge this gap, making information easily shareable between contexts and process phases to limit users' cognitive load. This is an area where symbolic execution is well suited and has been used in the fully-automated context [225]. However, this sharing and context swapping should

be clearly communicated to users, they should have control, and tools should support interactivity across contexts.

Provide input and output in the context of code (G5). Next, software analysis tools should support interaction directly in the context of code (N=17). First tools should present output in the IDE or with relevant code segments so users can see results' relation to the code they are working with. When investigating questions developers ask of analysis tools, Smith et al. found wanted to understand how a tool's output relates to the codebase [221]. Similarly, users should be able to easily switch from working with the code to running the analysis. This could mean targeting an analysis by highlighting a function of interest. When investigating code annotation's impact on developers' program understanding, Krüger et al. showed lightweight annotations improved program understanding without extensive training [129].

Support collaboration (G6). Finally, tools should support collaborative analysis (N=23). Many tools view the analysis itself as conducted independently, with only the analysis' results shared between collaborators. However, prior work suggests it is important to share information about the analysis (e.g., configuration settings, intermediate results, etc.). Piskachev et al. emphasized the need to share these configuration options with others [185]. Similarly, Sadowski et al. describe the importance of tools supporting intra-team collaboration to set team coding guidelines to be used in static analyses when designing security tools for Google [203]. Moving beyond developers, Saha et al. emphasized the importance of collaboration in their interviews with malware analysts investigating APTs [204]. Also, Yamagishi et al., find collaboration is important to malware analysts generally and outline the typical modes of collaboration currently used [271].

6 Applying Usability Guidelines to SE

Next, we demonstrate our modular framework's utility by using it to show how the DSP usability guidelines can be applied to SE. For brevity, we discuss changes by theme, as opposed to by-guideline, as some changes affect multiple guidelines. We describe each theme impacts relevant modules and guidelines in turn and summarize the set of modules requiring changes for each guideline in Table 7.

Improve readability of symbolic constraints. The most obvious concern for SE usability is to ensure the final and intermediate constraint statements produced by the constraint solver are understandable. SE constraints are often complex logic representations, which precisely describe the possible inputs. However, if these are simplified or are hard to parse, they can make the results uninformative (G3) and particularly challenging for beginners (G2). Further, sharing these complex constraints during collaboration is likely challenging as all parties attempt to come to a shared understanding (G6). While there is likely some room for technical improvement in our ability to automatically identify possible constraint simplifications, we do not expect all constraints will be simplifiable. Therefore, research is necessary to identify ways of presenting the constraints that is more readable.

The most obvious solution is to present these constraints in the context of code (G5). This limits mental context switching between the constraints and the code defining the constraints and helps users investigate why the constraints were produced by tracing back execution through the code. Other solutions include breaking up the constraints into smaller sub-constraints or presenting the stages of constraint development throughout the code. Both focus on constraint components and guide understanding. The former also provides code as context for constraint understanding.

Produce defaults, but allow customization for all modules. Next, we see a need to change all modules to ensure SE is easy to start, but allows expert user customization (G2). It is necessary to determine reasonable defaults for decisions across each component. For example, a tool might begin with a machine learning-based search heuristic like Luo et al's. [148]. This reduces beginners' initial effort, but it likely will introduce inaccuracies due to inherent model limitations. As users become more expert, the tool might provide the ability to write additional heuristics to pre-empt the machine learning-based approach, leveraging the users' experience directly. While many tools already do this to some extent, either the defaults are not well communicated or the modules are hard to customize without rewriting the tool. Both require deep tool implementation knowledge and are challenging as it is not clear what the ideal defaults should be. More research is needed to identify useful mechanisms for efficient customization.

Present vulnerability examples, code fixes, and limitations to produce informative results. To produce informative results (G3), SE tools must provide additional context about results. SE seems particularly well-equipped to provide this context by design as the final constraint inherently provides context about the crashing inputs. One commonly discussed mechanism for producing informative results in DSP papers was through example inputs to demonstrate the vulnerability. This is already possible in the constraint solver module as it requires identifying an input meeting the constraint. Similarly, many DSP papers recommended providing suggested fixes simplify developers' vulnerability triage process. While this is more complicated, the constrain solver could be used to identify conditions that would need to be changed to invalidate the constraint. Then, we would modify the execution driver to operate in reverse or maintain a history of the symbolic execution trace to identify the instruction(s) producing the target conditions. Once the appropriate instruction(s) is identified, suggested edits could be generated which invalidate the vulnerable constraint.

In addition to providing crash information, the DSP literature suggests tools describe their own limitations. Changes to SE tools should be made to somewhat reveal the environment and memory model, as well as the search heuristic, which impacted the final constraint set. For example, if the final constraint included elements of a simplified memory structure representation, this should be flagged so users could decide whether to investigate the impact of a more robust memory representation on the analysis result.

Allow easy heuristic and environment tweaking. To improve SE interactivity (G4) requires considering the search heuristic and environmental model. These modules offer clear opportunities for control of the SE engine as the user learns more about the program. As the search heuristic pushes execution down different program paths, user could revert to a prior state and revise the path decision or drive the decision interactively at the outset. Similarly, as the user learns more about the program, and its interaction with the environment and memory, they might decide they

Usability Guideline	Paper count	Execution Driver	Search Heuristic	Constraint Solver	Env. & Mem. Model	Sym. State Manager
Integrate with workflow (G1)	38	X		X		
Easy to get started, but ability to go deep (G2)	29	X	X	X	X	X
Informative results are critical (G3)	47	X	X	X	X	
Interactive use; with human in tight loop (G4)	21	X	X		X	X
Provide input and output in the context of code (G5)	14		X	X	X	
Support collaboration (G6)	22		X	X	X	

Table 7: Usability Guideline Mapping

need to change representations. Being able to quickly modify these symbolic representations, possibly for a single variable, without significantly rewriting the code would likely improve usability. Both modifications should be presented in the context of the code (G5) to allow users to make interaction decisions while viewing program elements. This suggests a GUI-based interface, such as angr-management [1], might be most appropriate as it allows users to step through SE for each line and make heuristic decisions and changes to the environmental model. Conversely, we do not believe changes to the execution driver, constraint solver, or symbolic state manager are needed for interactivity. These modules are less dependent on the program specifics and therefore offer fewer decision points at which users would drive the tool.

Add the ability to share configuration info. To best support collaboration (G6), we recommend tool developers focus on approaches to share search heuristics and environmental and memory models. As discussed above, these modules offer the most user decision points in the process. Therefore, they represent the most pressing point to ensure consistency of use between collaborators, as well as the clearest opportunity to share insights while investigating programs. Sharing configurations for the other modules is also important, but because they are likely to be modified less frequently, do not present as significant a challenge for consistency. That is, any configuration changes can likely be shared manually without as much need for tool support.

Integrating with DSP users' workflows requires tailoring and interactive controls. Finally, we turn to our initial guideline of integrating into DSP users' workflows (G1). Because workflows can vary between user groups, meeting this guideline requires a tailored approach across many modules. For example, a major challenge in software reverse engineering is transitioning between static and dynamic contexts between analysis phases. SE already seems well situated to resolve this issue by leveraging identified constraints to produce a set of inputs to guide execution. It also seems likely that pairing a similar approach to bridge software analysis and network packet analysis could be useful for protocol reverse engineering, presenting constraints identified for network calls in the context of captured packet data. However, other changes would be required to integrate SE with earlier design aspects of software developers processes, such as requirements development and threat modeling. These processes are not grounded as directly in code, so other modules would need to be changed to allow SE tools to capture the results of these phases and utilize their output. For example, once a developer produces a requirement, this could inform the environment and memory model for a particular feature test. Also, the threat model could be used to evaluate the SE tool's final constraints to determine whether a vulnerability is possible.

The other area of workflow integration from the DSP literature was the importance of presenting analysis results at the right time and level of detail. For SE, this suggests it is important to allow intermittent execution in the execution driver and present possible constraints as the user moves through a program. If the user has to wait for constraints to be produced after executing over a full function or larger amount of code and only for a target address, this limits their ability to receive feedback at the right time. However, this information presentation should be controllable by the user, otherwise it risks potential information overload.

7 Conclusion and Future Work

In this paper, we present a systemization of knowledge that tackles the problem of how to foster the current and future symbolic execution research by creating a flexible, research-friendly SE framework with proper usability. We categorize existing SE advancements and applications and organize them under their respective SE modules with specific design requirements. We also demonstrate how our modules can be used to drive future SE research by focusing on SE usability. Based on the qualitative study results, we provide both technical and usability guidelines for the design and development of such a SE framework, however, we note that these guidelines should be validated through future user studies.

Acknowledgments

This work is sponsored by, and related to, Department of Navy award N00014-23-1-2563 issued by the Office of Naval Research, the Defense Advanced Research Projects Agency (DARPA) agreement number FA875019C0003, and the National Science Foundation (NSF) grants 2247959, 2247954, and 2442984. It is also based on work supported by the Advanced Research Projects Agency for Health (ARPA-H) under Contract No. SP4701-23-C-0074. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of any of these funding organizations.

References

[1] 2025. angr-management: The official angr GUI. https://github.com/angr/angr-management.
[2] 2026. KLEE Symbolic Execution Engine. http://klee.github.io/.

[3] Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L Mazurek, and Christian Stransky. 2017. Comparing the Usability of Cryptographic APIs. In *IEEE Symposium on Security and Privacy (S&P)*.

[4] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L Mazurek, and Christian Stransky. 2016. You Get Where You're Looking for: The Impact of Information Sources on Code Security. In *IEEE Symposium on Security and Privacy (S&P)*.

[5] Ashish Aggarwal and Pankaj Jalote. 2006. Integrating Static and Dynamic Analysis for Detecting Vulnerabilities. In *Annual International Computer Software and Applications Conference (COMPSAC)*. IEEE.

[6] Reza Ahmadi and Juergen Dingel. 2019. Concolic Testing for Models of State-based Systems. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[7] Mujtahid Akon, Tianchang Yang, Yilu Dong, and Syed Rafiul Hussain. 2023. Formal Analysis of Access Control Mechanism of 5G Core Network. In *ACM Conference on Computer and Communications Security (CCS)*.

[8] Fritz Alder, Lesly-Ann Daniel, David Oswald, Frank Piessens, and Jo Van Bulck. 2024. Pandora: Principled Symbolic Validation of Intel SGX Enclave Runtimes. In *IEEE Symposium on Security and Privacy (S&P)*.

[9] Maysara Alhindi and Joseph Hallett. 2025. Playing in the Sandbox: A Study on the Usability of Seccomp. *arXiv preprint arXiv:2506.10234* (2025).

[10] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and V N Venkatakrishnan. 2018. NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications. In *USENIX Security Symposium*.

[11] Noura Alomar, Primal Wijesekera, Edward Qiu, and Serge Egelman. 2020. "You've got your nice list of bugs, now what?" Vulnerability Discovery and Management Processes in the Wild. In *Symposium on Usable Privacy and Security (SOUPS)*.

[12] Omar Alrawi, Moses Ike, Matthew Pruett, Ranjita Pai Kasturi, Srimanta Barua, Taleb Hirani, Brennan Hill, and Brendan Saltaformaggio. 2021. Forecasting Malware Capabilities From Cyber Attack Memory Images. In *USENIX Security Symposium*.

[13] Amit Seal Ami, Nathan Cooper, Kaushal Kafle, Kevin Moran, Denys Poshyvanyk, and Adwait Nadkarni. 2022. Why Crypto-Detectors Fail: A Systematic Evaluation of Cryptographic Misuse Detection Techniques. In *IEEE Symposium on Security and Privacy (S&P)*.

[14] Amit Seal Ami, Kevin Moran, Denys Poshyvanyk, and Adwait Nadkarni. 2024. False negative—that one is going to kill you": Understanding Industry Perspectives of Static Analysis based Security Testing. In *IEEE Symposium on Security and Privacy (S&P)*.

[15] Andrea Aquino, Giovanni Denaro, and Mauro Pezze. 2017. Heuristically Matching Solution Spaces of Arithmetic Formulas to Efficiently Reuse Solutions. In *International Conference on Software Engineering (ICSE)*.

[16] Shanai Ardi, David Byers, Per Hakon Meland, Inger Anne Tondel, and Nahid Shahmehri. 2007. How Can the Developer Benefit from Security Modeling?. In *International Conference on Availability, Reliability and Security (ARES)*.

[17] Hala Assal and Sonia Chiasson. 2018. Security in the Software Development Lifecycle. In *Symposium on Usable Privacy and Security (SOUPS)*.

[18] Vytautas Astrauskas, Aurel Bílý, Jonáš Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. 2022. The Prusti Project: Formal Verification for Rust. In *NASA Formal Methods*. Vol. 13260. Springer International Publishing, 88–108. doi:10.1007/978-3-031-06773-0_5 Series Title: Lecture Notes in Computer Science.

[19] Thanassis Avgerinos, Sang Kil Cha, Brent Tze Hao Lim, and David Brumley. 2011. AEG: Automatic Exploit Generation. In *Symposium on Network and Distributed System Security (NDSS)*.

[20] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing Symbolic Execution with Veritesting. In *International Conference on Software Engineering (ICSE)*.

[21] Nathaniel Ayewah, David Hovemeyer, J. Morgenthaler, John Penix, and William Pugh. 2008. Using Static Analysis to Find Bugs. *IEEE Software* 25 (2008), 22–29. doi:10.1109/MS.2008.130

[22] Kushal Babel, Philip Daian, Mahimna Kelkar, and Ari Juels. 2023. Clockwork Finance: Automated Analysis of Economic Security in Smart Contracts. In *IEEE Symposium on Security and Privacy (S&P)*.

[23] Joshua Bailey and Charles Nicholas. 2025. Symbolic Execution in Practice: A Survey of Applications in Vulnerability, Malware, Firmware, and Protocol Analysis. *arXiv preprint arXiv:2508.06643* (2025).

[24] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *Comput. Surveys* 51, 3, Article 50 (2018), 39 pages. doi:10.1145/3182657

[25] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. 2006. Thorough Static Analysis of Device Drivers. *ACM SIGOPS Operating Systems Review* 40, 4 (2006).

[26] Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, and David Brumley. 2017. Your Exploit is Mine: Automatic Shellcode Transplant for Remote Exploits. In *IEEE Symposium on Security and Privacy (S&P)*.

[27] Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*.

[28] Steffen Becker, Carina Wiesen, Nils Albartus, Nikol Rummel, and Christof Paar. 2020. An Exploratory Study of Hardware Reverse Engineering – Technical and Cognitive Processes. In *Symposium on Usable Privacy and Security (SOUPS)*.

[29] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. 2018. On the Dichotomy of Debugging Behavior among Programmers. In *International Conference on Software Engineering (ICSE)*.

[30] Luca Borzacchiello, Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. 2019. Memory Models in Symbolic Execution: Key Ideas and New Thoughts. *Software Testing, Verification and Reliability* 29, 8 (2019), e1722.

[31] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. 2022. Handling Memory-Intensive Operations in Symbolic Execution. In *Innovations in Software Engineering Conference*. ACM.

[32] Luca Borzacchiello, Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. 2019. Reconstructing C2 Servers for Remote Access Trojans with Symbolic Execution. In *Cyber Security, Cryptography and Machine Learning (Lecture Notes in Computer Science)*. Springer International Publishing.

[33] Tegan Brennan, Seemanta Saha, Tevfik Bultan, and Corina S. Păsăreanu. 2018. Symbolic Path Cost Analysis for Side-channel Detection. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.

[34] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut Kandemir. 2019. CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation. In *IEEE Symposium on Security and Privacy (S&P)*.

[35] Fraser Brown, Deian Stefan, and Dawson Engler. 2020. Sys: A Static/Symbolic Tool for Finding Good Bugs in Good (Browser) Code. In *USENIX Security Symposium*.

[36] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. 2011. Parallel Symbolic Execution for Automated Real-world Software Testing. In *European Conference on Computer Systems (EuroSys)*. ACM.

[37] Suhabe Bugrara and Dawson Engler. 2013. Redundant State Detection for Dynamic Symbolic Execution. In *USENIX Annual Technical Conference (ATC)*.

[38] Frank Busse, Martin Nowack, and Cristian Cadar. 2020. Running Symbolic Execution Forever. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.

[39] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. 2011. Symbolic Execution for Software Testing in Practice. In *International Conference on Software Engineering (ICSE)*.

[40] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2 (2013), 82–90.

[41] Dustin Campbell and Mark Miller. 2008. Designing Refactoring Tools for Developers. In *Workshop on Refactoring Tools*.

[42] Chen Cao, Le Guan, Jiang Ming, and Peng Liu. 2020. Device-agnostic Firmware Execution is Possible: A Concolic Execution Approach for Peripheral Emulation. In *Annual Computer Security Applications Conference (ACSAC)*.

[43] Sooyoung Cha, Seongjoon Hong, Jingyoung Kim, Junhee Lee, and Hakjoo Oh. 2022. Enhancing Dynamic Symbolic Execution by Automatically Learning Search Heuristics. *IEEE Transactions on Software Engineering* 48, 9 (2022), 3640–3663.

[44] Sooyoung Cha, Seonho Lee, and Hakjoo Oh. 2018. Template-guided Concolic Testing via Online Learning. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*.

[45] Sooyoung Cha and Hakjoo Oh. 2019. Concolic Testing with Adaptively Changing Search Heuristics. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[46] Sooyoung Cha and Hakjoo Oh. 2020. Making Symbolic Execution Promising by Learning Aggressive State-pruning Strategy. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[47] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *IEEE Symposium on Security and Privacy (S&P)*.

[48] Stephen Chang, Alex Knauth, and Emina Torlak. 2017. Symbolic Types for Lenient Symbolic Execution. *ACM on Programming Languages* 2, POPL (2017), 40:1–40:29.

[49] Kathy Charmaz. 2006. *Constructing Grounded Theory: A Practical Guide through Qualitative Analysis*. Sage.

[50] Sze Yiu Chau, Omar Chowdhury, Endadul Hoque, Huangyi Ge, Aniket Kate, Cristina Nita-Rotaru, and Ninghui Li. 2017. SymCerts: Practical Symbolic Execution for Exposing Noncompliance in X.509 Certificate Validation Implementations. In *IEEE Symposium on Security and Privacy (S&P)*.

[51] Sze Yiu Chau, Moosa Yahyazadeh, Omar Chowdhury, Aniket Kate, and Ninghui Li. 2019. Analyzing Semantic Correctness with Symbolic Execution: A Case Study on PKCS#1 v1.5 Signature Verification. In *Symposium on Network and Distributed System Security (NDSS)*.

[52] Thierry Titcheu Chekam, Mike Papadakis, Maxime Cordy, and Yves Le Traon. 2021. Killing Stubborn Mutants with Symbolic Execution. *ACM Transactions on Software Engineering and Methodology* 30, 2 (2021).

[53] Bo Chen, Christopher Havlicek, Zhenkun Yang, Kai Cong, Raghudeep Kannavara, and Fei Xie. 2018. CRETE: A Versatile Binary-Level Concolic Testing Framework. In *Fundamental Approaches to Software Engineering.* Springer International Publishing.

[54] Ju Chen, Wookhyun Han, Mingjun Yin, Haochen Zeng, Chengyu Song, Byoungyoung Lee, Heng Yin, and Insik Shin. 2022. SYMSAN: Time and Space Efficient Concolic Execution via Dynamic Data-flow Analysis. In *USENIX Security Symposium.*

[55] Jianhui Chen and Fei He. 2018. Control Flow-guided SMT Solving for Program Verification. In *ACM/IEEE International Conference on Automated Software Engineering (ASE).*

[56] Ju Chen, Jinghan Wang, Chengyu Song, and Heng Yin. 2022. JIGSAW: Efficient and Scalable Path Constraints Fuzzing. In *IEEE Symposium on Security and Privacy (S&P).*

[57] Ting Chen, Xiaodong Lin, Jin Huang, Abel Bacchus, and Xiaosong Zhang. 2015. An Empirical Investigation into Path Divergences for Concolic Execution using CREST. *Security and Communication Networks* 8, 18 (2015), 3667–3681.

[58] Weiteng Chen, Yu Hao, Zheng Zhang, Xiaochen Zou, Dhilung Kirat, Shachee Mishra, Douglas Schales, Jiyong Jang, and Zhiyun Qian. 2024. SyzGen++: Dependency Inference for Augmenting Kernel Driver Fuzzing. In *IEEE Symposium on Security and Privacy (S&P).*

[59] Weimin Chen, Zihan Sun, Haoyu Wang, Xiapu Luo, Haipeng Cai, and Lei Wu. 2022. WASAI: Uncovering Vulnerabilities in Wasm Smart Contracts. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).*

[60] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Taowei, and Long Lu. 2020. SAVIOR: Towards Bug-Driven Hybrid Testing. In *IEEE Symposium on Security and Privacy (S&P).*

[61] Zhenbang Chen, Zehua Chen, Ziqi Shuai, Guofeng Zhang, Weiyu Pan, Yufeng Zhang, and Ji Wang. 2021. Synthesize Solving Strategy for Symbolic Execution. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).*

[62] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2012. The S2E Platform: Design, Implementation, and Applications. *ACM Transactions on Computer Systems* 30, 1 (2012), 1–49.

[63] Mingi Cho, Seoyoung Kim, and Taekyoung Kwon. 2019. Intriguer: Field-Level Constraint Solving for Hybrid Fuzzing. In *ACM Conference on Computer and Communications Security (CCS).*

[64] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. 2019. Grey-Box Concolic Testing on Binary Code. In *International Conference on Software Engineering (ICSE).*

[65] Maria Christakis and Christian Bird. 2016. What Developers Want and Need from Program Analysis: An Empirical Study. In *ACM/IEEE International Conference on Automated Software Engineering (ASE).*

[66] Maria Christakis, Peter Müller, and Valentin Wüstholz. 2016. Guiding Dynamic Symbolic Execution toward Unverified Program Executions. In *International Conference on Software Engineering (ICSE).*

[67] Tobias Cloosters, Michael Rodler, and Lucas Davi. 2020. TeeRex: Discovery and Exploitation of Memory Corruption Vulnerabilities in SGX Enclaves. In *USENIX Security Symposium.*

[68] Michael Coblenz, Gauri Kambhatla, Paulette Koronkevich, Jenna L Wise, Celeste Barnaby, Joshua Sunshine, Jonathan Aldrich, and Brad A Myers. 2021. PLIERS: A Process that Integrates User-Centered Methods into Programming Language Design. *ACM Transactions on Computer-Human Interaction (TOCHI)* 28, 4 (2021), 1–53.

[69] Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. 2017. Rethinking Pointer Reasoning in Symbolic Execution. In *ACM/IEEE International Conference on Automated Software Engineering (ASE).*

[70] Emilio Coppa, Heng Yin, and Camil Demetrescu. 2022. SymFusion: Hybrid Instrumentation for Concolic Execution. In *ACM/IEEE International Conference on Automated Software Engineering (ASE).*

[71] Nassim Corteggiani, Giovanni Camurati, and Aurelien Francillon. 2018. Inception: System-Wide Security Testing of Real-World Embedded Systems Software. In *USENIX Security Symposium.*

[72] Lesly-Ann Daniel, Sebastien Bardin, and Tamara Rezk. 2020. Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level. In *IEEE Symposium on Security and Privacy (S&P).*

[73] Anastasia Danilova, Alena Naiakshina, and Matthew Smith. 2020. One Size Does Not Fit All: A Grounded Theory and Online Survey Study of Developer Preferences for Security Warning Types. In *International Conference on Software Engineering (ICSE).*

[74] Jens Dietrich, Shawn Rasheed, Alexander Jordan, and Tim White. 2023. On the Security Blind Spots of Software Composition Analysis. In *Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses.*

[75] Sushant Dinesh, Madhusudan Parthasarathy, and Christopher W Fletcher. 2024. Conjunct: Learning Inductive Invariants to Prove Unbounded Instruction Safety against Microarchitectural Timing Attacks. In *IEEE Symposium on Security and Privacy (S&P).*

[76] Jenna DiVincenzo, Ian McCormack, Conrad Zimmerman, Hemant Gouni, Jacob Gorenburg, Jan-Paul Ramos-Dávila, Mona Zhang, Joshua Sunshine, Éric Tanter, and Jonathan Aldrich. 2025. Gradual C0: Symbolic Execution for Gradual Verification. *ACM Transactions on Programming Languages and Systems* 46, 4, Article 14 (2025), 57 pages.

[77] Lisa Nguyen Quang Do, James R Wright, and Karim Ali. 2020. Why Do Software Developers Use Static Analysis Tools? A User-centered Study of Developer Needs and Motivations. *IEEE Transactions on Software Engineering* 48, 3 (2020), 835–847.

[78] T.A. Do, Alvis Fong, and R. Pears. 2012. Dynamic Symbolic Execution Guided by Data Dependency Analysis for High Structural Coverage. In *International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE).*

[79] Will Epperson, Gagan Bansal, Victor C Dibia, Adam Fourney, Jack Gerrits, Erkang Zhu, and Saleema Amershi. 2025. Interactive Debugging and Steering of Multi-Agent AI Systems. In *ACM CHI Conference on Human Factors in Computing Systems (CHI).*

[80] Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. 2013. Rethinking SSL Development in an Appified World. In *ACM Conference on Computer and Communications Security (CCS).*

[81] Kostas Ferles, Jacob Van Geffen, Isil Dillig, and Yannis Smaragdakis. 2018. Symbolic Reasoning for Automatic Signal Placement. *SIGPLAN Notices* 53, 4 (2018), 120–134. doi:10.1145/3296979.3192395

[82] Irina Ford, Ananta Soneji, Faris Bugra Kokulu, Jayakrishna Vadayath, Zion Leonahenahe Basque, Gaurav Vipat, Adam Doupé, Ruoyu Wang, Gail-Joon Ahn, Tiffany Bao, et al. 2024. "Watching over the shoulder of a professional": Why Hackers Make Mistakes and How They Fix Them. In *IEEE Symposium on Security and Privacy (S&P).*

[83] José Fragoso Santos, Petar Maksimović, Sacha-Elie Ayoun, and Philippa Gardner. 2020. Gillian, Part I: A Multi-language Platform for Symbolic Execution. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

[84] Joel Frank, Cornelius Aschermann, and Thorsten Holz. 2020. ETHBMC: A Bounded Model Checker for Smart Contracts. In *USENIX Security Symposium.*

[85] Sneha Gathani, Peter Lim, and Leilani Battle. 2020. Debugging Database Queries: A Survey of Tools, Techniques, and Users. In *ACM CHI Conference on Human Factors in Computing Systems (CHI).*

[86] Lisa Geierhaas, Anna-Marie Ortloff, Matthew Smith, and Alena Naiakshina. 2022. Let's Hash: Helping Developers with Password Security. In *Symposium on Usable Privacy and Security (SOUPS).*

[87] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI).*

[88] Peter Leo Gorski, Yasemin Acar, Luigi Lo Iacono, and Sascha Fahl. 2020. Listen to Developers! A Participatory Design Study on Security Warnings for Cryptographic APIs. In *ACM CHI Conference on Human Factors in Computing Systems (CHI).*

[89] Peter Leo Gorski, Luigi Lo Iacono, Dominik Wermke, Christian Stransky, Sebastian Möller, Yasemin Acar, and Sascha Fahl. 2018. Developers Deserve Security Warnings, Too: On the Effect of Integrated Security Advice on Cryptographic API Misuse. In *Symposium on Usable Privacy and Security (SOUPS).*

[90] Michaela Greiler, Margaret-Anne Storey, and Abi Noda. 2022. An Actionable Framework for Understanding and Improving Developer Experience. *IEEE Transactions on Software Engineering* 49, 4 (2022), 1411–1425.

[91] Fabio Gritti, Lorenzo Fontana, Eric Gustafson, Fabio Pagani, Andrea Continella, Christopher Kruegel, and Giovanni Vigna. 2020. SYMBION: Interleaving Symbolic with Concrete Execution. In *IEEE Conference on Communications and Network Security (CNS).*

[92] Fabio Gritti, Lorenzo Fontana, Eric Gustafson, Fabio Pagani, Andrea Continella, Christopher Kruegel, and Giovanni Vigna. 2020. SYMBION: Interleaving Symbolic with Concrete Execution. In *Proceedings of the IEEE Conference on Communications and Network Security (CNS).*

[93] Hui Guo and Cindy Rubio-González. 2020. Efficient Generation of Error-inducing Floating-point Inputs via Symbolic Execution. In *International Conference on Software Engineering (ICSE).*

[94] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. 2020. SpecuSym: Speculative Symbolic Execution for Cache Timing Leak Detection. In *International Conference on Software Engineering (ICSE).*

[95] Shengjian Guo, Meng Wu, and Chao Wang. 2018. Adversarial Symbolic Execution for Detecting Concurrency-Related Cache Timing Leaks. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).*

[96] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge Navas. 2015. The SeaHorn Verification Framework. In *International Conference on Computer-Aided Verification (CAV).*

[97] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. 2013. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *USENIX Security Symposium*.

[98] HyungSeok Han, JeongOh Kyea, Yonghwi Jin, Jinoh Kang, Brian Pak, and Insu Yun. 2023. QueryX: Symbolic Query on Decompiled Code for Finding Bugs in COTS Binaries. In *IEEE Symposium on Security and Privacy (S&P)*.

[99] Ahmed E Hassan, Gustavo A Oliva, Dayi Lin, Boyuan Chen, Zhen Ming, et al. 2024. Rethinking Software Engineering in the Foundation Model Era: From Task-driven AI Copilots to Goal-driven AI Pair Programmers. *arXiv preprint arXiv:2404.10225* (2024).

[100] Sk Adnan Hassan, Zainab Aamir, Dongyoon Lee, James C Davis, and Francisco Servant. 2023. Improving Developers' Understanding of Regex Denial of Service Tools through Anti-Patterns and Fix Strategies. In *IEEE Symposium on Security and Privacy (S&P)*.

[101] Andrew F Hayes and Klaus Krippendorff. 2007. Answering the Call for a Standard Reliability Measure for Coding Data. *Communication Methods and Measures* 1, 1 (2007), 77–89.

[102] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to Fuzz from Symbolic Execution with Application to Smart Contracts. In *ACM Conference on Computer and Communications Security (CCS)*.

[103] Weigang He, Peng Di, Mengli Ming, Chengyu Zhang, Ting Su, Shijie Li, and Yulei Sui. 2024. Finding and Understanding Defects in Static Analyzers by Constructing Automated Oracles. *ACM on Software Engineering* 1, FSE (2024), 1656–1678.

[104] Dulaji Hidellaarachchi, John Grundy, Rashina Hoda, and Ingo Mueller. 2023. The Influence of Human Aspects on Requirements Engineering-related Activities: Software Practitioners' Perspective. *ACM Transactions on Software Engineering and Methodology* 32, 5 (2023), 1–37.

[105] Sandra Höltervennhoff, Philip Klostermeyer, Noah Wöhler, Yasemin Acar, and Sascha Fahl. 2023. "I wouldn't want my unsafe code to run my pacemaker": An Interview Study on the Use, Comprehension, and Perceived Risks of Unsafe Rust. In *USENIX Security Symposium*.

[106] Jie Hu, Yue Duan, and Heng Yin. 2024. Marco: A Stochastic Asynchronous Concolic Explorer. In *International Conference on Software Engineering (ICSE)*.

[107] Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. 2020. Pangolin: Incremental Hybrid Fuzzing with Polyhedral Path Abstraction. In *IEEE Symposium on Security and Privacy (S&P)*.

[108] Jeff Huang and Lawrence Rauchwerger. 2015. Finding Schedule-sensitive Branches. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[109] Kaiming Huang, Mathias Payer, Zhiyun Qian, Jack Sampson, Gang Tan, and Trent Jaeger. 2024. Top of the Heap: Efficient Memory Error Protection of Safe Heap Objects. In *ACM Conference on Computer and Communications Security (CCS)*.

[110] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. 2022. "They're not that hard to mitigate": What Cryptographic Library Developers Think about Timing Attacks. In *IEEE Symposium on Security and Privacy (S&P)*.

[111] Xiangyang Jia, Carlo Ghezzi, and Shi Ying. 2015. Enhancing Reuse of Constraint Solutions to Improve Symbolic Execution. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.

[112] Zheyue Jiang, Yuan Zhang, Jun Xu, Qi Wen, Zhenghe Wang, Xiaohan Zhang, Xinyu Xing, Min Yang, and Zhemin Yang. 2020. PDiff: Semantic-based Patch Presence Testing for Downstream Kernels. In *ACM Conference on Computer and Communications Security (CCS)*.

[113] Ling Jin, Yinzhi Cao, Yan Chen, Di Zhang, and Simone Campanoni. 2023. ExGen: Cross-platform, Automated Exploit Generation for Smart Contract Vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* 20, 1 (2023).

[114] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *International Conference on Software Engineering (ICSE)*.

[115] Evan Johnson, Maxwell Bland, YiFei Zhu, Joshua Mason, Stephen Checkoway, Stefan Savage, and Kirill Levchenko. 2021. Jetset: Targeted Firmware Rehosting for Embedded Systems. In *USENIX Security Symposium*.

[116] Timotej Kapus, Frank Busse, and Cristian Cadar. 2020. Pending Constraints in Symbolic Execution for Better Exploration and Seeding. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*.

[117] Timotej Kapus and Cristian Cadar. 2019. A Segmented Memory Model for Symbolic Execution. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[118] Samantha Katcher, James Mattei, Jared Chandler, and Daniel Votipka. 2025. An Investigation of Interaction and Information Needs for Protocol Reverse Engineering Automation. In *ACM CHI Conference on Human Factors in Computing Systems (CHI)*.

[119] Mannat Kaur, Michel Van Eeten, Marijn Janssen, Kevin Borgolte, and Tobias Fiebig. 2021. Human Factors in Security Research: Lessons Learned from 2008-2018. *arXiv preprint arXiv:2103.13287* (2021).

[120] Dilara Keküllüoğlu and Yasemin Acar. 2023. "We are a startup to the core": A qualitative interview study on the security and privacy development practices in Turkish software startups. In *IEEE Symposium on Security and Privacy (S&P)*.

[121] Steven Keuchel, Sander Huyghebaert, Georgy Lukyanov, and Dominique Devriese. 2022. Verified Symbolic Execution with Kripke Specification Monads (and no Meta-programming). *ACM on Programming Languages* 6, ICFP (2022).

[122] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. 2003. Generalized Symbolic Execution for Model Checking and Testing. In *Symposium on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

[123] Kyungtae Kim, Dae R. Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel. In *Symposium on Network and Distributed System Security (NDSS)*.

[124] James C King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (1976), 385–394.

[125] Alexandra Klymenko, Stephen Meisenbacher, Patrick Gage Kelley, Sai Teja Peddinti, Kurt Thomas, and Florian Matthes. 2025. "We are not Future-ready": Understanding AI Privacy Risks and Existing Mitigation Strategies from the Perspective of AI Developers in Europe. In *Symposium on Usable Privacy and Security (SOUPS)*.

[126] Alexander Krause, Harjot Kaur, Jan H Klemmer, Oliver Wiese, and Sascha Fahl. 2025. "That's my perspective from 30 years of doing this": An Interview Study on Practices, Experiences, and Challenges of Updating Cryptographic Code. In *USENIX Security Symposium*.

[127] Alexander Krause, Jan H Klemmer, Nicolas Huaman, Dominik Wermke, Yasemin Acar, and Sascha Fahl. 2023. Pushed by Accident: A Mixed-Methods Study on Strategies of Handling Secret Information in Source Code Repositories. In *USENIX Security Symposium*.

[128] Daniel Kroening, Peter Schrammel, and Michael Tautschnig. 2023. CBMC: The C Bounded Model Checker. http://arxiv.org/abs/2302.02384 arXiv:2302.02384 [cs].

[129] Jacob Krüger, Gül Çalıklı, Thorsten Berger, Thomas Leich, and Gunter Saake. 2019. Effects of Explicit Feature Traceability on Program Comprehension. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[130] Johannes Krupp and Christian Rossow. 2018. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *USENIX Security Symposium*.

[131] Daniil Kuts. 2021. Towards Symbolic Pointers Reasoning in Dynamic Symbolic Execution. In *Ivannikov Memorial Workshop (IVMEM)*. IEEE.

[132] Heng Li, Weiyi Shang, Bram Adams, Mohammed Sayagh, and Ahmed E Hassan. 2020. A Qualitative Study of the Benefits and Costs of Logging from Developers' Perspectives. *IEEE Transactions on Software Engineering* 47, 12 (2020), 2858–2873.

[133] Penghui Li, Wei Meng, Kangjie Lu, and Changhua Luo. 2021. On the Feasibility of Automated Built-in Function Modeling for PHP Symbolic Execution. In *The Web Conference*. ACM.

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

[135] Xin Li, Yongjuan Liang, Hong Qian, Yi-Qi Hu, Lei Bu, Yang Yu, Xin Chen, and Xuandong Li. 2016. Symbolic Execution of Complex Program Driven by Machine Learning based Constraint Solving. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*.

[136] You Li, Zhendong Su, Linzhang Wang, and Xuandong Li. 2013. Steering Symbolic Execution to Less Traveled Paths. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*.

[137] Hongliang Liang, Wenqing Yu, Lu Ai, and Lin Jiang. 2020. A Practical Concolic Execution Technique for Large Scale Software Systems. In *Evaluation and Assessment in Software Engineering*. ACM.

[138] Jenny T. Liang, Maryam Arab, Minhyuk Ko, Amy J. Ko, and Thomas D. LaToza. 2023. A Qualitative Study on the Implementation Design Decisions of Developers. In *International Conference on Software Engineering (ICSE)*.

[139] Zhengchuan Liang, Xiaochen Zou, Chengyu Song, and Zhiyun Qian. 2024. K-LEAK: Towards Automating the Generation of Multi-Step Infoleak Exploits against the Linux Kernel. In *Symposium on Network and Distributed System Security (NDSS)*.

[140] Elizabeth Lin, Sparsha Gowda, William Enck, and Dominik Wermke. 2025. Context Matters: Qualitative Insights into Developers' Approaches and Challenges with Software Composition Analysis. In *USENIX Security Symposium*.

[141] Amanda Liu and Michael Coblenz. 2023. Debugging Techniques in Professional Programming. In *Plateau Workshop*.

[142] Dongge Liu, Gidon Ernst, Toby Murray, and Benjamin I. P. Rubinstein. 2020. Legion: Best-first Concolic Testing. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*.

[143] Yingtong Liu, Hsin-Wei Hung, and Ardalan Amiri Sani. 2020. Mousse: A System for Selective Symbolic Execution of Programs with Untamed Environments. In *European Conference on Computer Systems (EuroSys)*. ACM.

[144] Zhengyu Liu, Kecheng An, and Yinzhi Cao. 2024. Undefined-oriented Programming: Detecting and Chaining Prototype Pollution Gadgets in Node.js Template

Engines for Malicious Consequences. In *IEEE Symposium on Security and Privacy (S&P)*.

[145] Blake Loring, Duncan Mitchell, and Johannes Kinder. 2019. Sound Regular Expression Semantics for Dynamic Symbolic Execution of JavaScript. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[146] Changhua Luo, Penghui Li, Wei Meng, and Chao Zhang. 2024. Test Suites Guided Vulnerability Validation for Node.js Applications. In *ACM Conference on Computer and Communications Security (CCS)*.

[147] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. 2014. Semantics-based Obfuscation-resilient Binary Code Similarity Comparison with Applications to Software Plagiarism Detection. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[148] Sicheng Luo, Hui Xu, Yanxiang Bi, Xin Wang, and Yangfan Zhou. 2021. Boosting Symbolic Execution via Constraint Solving Time Prediction (Experience Paper). In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.

[149] Nuno Machado, Brandon Lucia, and Luís Rodrigues. 2015. Concurrency Debugging with Differential Schedule Projections. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[150] Alessandro Mantovani, Simone Aonzo, Yanick Fratantonio, and Davide Balzarotti. 2022. RE-Mind: A First Look inside the Mind of a Reverse Engineer. In *USENIX Security Symposium*.

[151] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: High-coverage Testing of Software Patches. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

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

[153] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. 2019. Reliability and Inter-rater Reliability in Qualitative Research: Norms and Guidelines for CSCW and HCI Practice. *ACM on Human-Computer Interaction* 3, CSCW (2019), 1–23.

[154] Sergey Mechtaev, Alberto Griggio, Alessandro Cimatti, and Abhik Roychoudhury. 2018. Symbolic Execution with Existential Second-order Constraints. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[155] Marcos Medeiros, Uirá Kulesza, Roberta Coelho, Rodrigo Bonifácio, Christoph Treude, and Eiji Adachi Barbosa. 2024. The Impact Of Bug Localization Based on Crash Report Mining: A Developers' Perspective. In *International Conference on Software Engineering: Software Engineering in Practice*.

[156] Xianya Mi, Sanjay Rawat, Cristiano Giuffrida, and Herbert Bos. 2021. LeanSym: Efficient Hybrid Fuzzing Through Conservative Constraint Debloating. In *International Symposium on Research in Attacks, Intrusions and Defenses*. ACM.

[157] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. 2017. BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking. In *USENIX Security Symposium*.

[158] Jaron Mink, Hadjer Benkraouda, Limin Yang, Arridhana Ciptadi, Ali Ahmadzadeh, Daniel Votipka, and Gang Wang. 2023. Everybody's Got ML, Tell Me What Else You Have: Practitioners' Perception of ML-Based Security Tools and Explanations. In *IEEE Symposium on Security and Privacy (S&P)*.

[159] Azadeh Mokhberi and Konstantin Beznosov. 2021. SoK: Human, Organizational, and Technological Dimensions of Developers' Challenges in Engineering Secure Software. In *European Symposium on Usable Security*.

[160] 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 *ACM/IEEE International Conference on Automated Software Engineering (ASE)*.

[161] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. 2018. Avatar^2: A Multi-Target Orchestration Platform. In *Workshop on Binary Analysis Research (BAR)*.

[162] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution. In *International Conference on Computer-Aided Verification (CAV)*. Springer International Publishing.

[163] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*.

[164] Brad A. Myers, Amy J. Ko, Thomas D. LaToza, and YoungSeok Yoon. 2016. Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools. *IEEE Computer* 49, 07 (2016).

[165] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an LLM to Help with Code Understanding. In *International Conference on Software Engineering (ICSE)*.

[166] Faezeh Nasrabadi, Robert Künnemann, and Hamed Nemati. 2023. CryptoBap: A Binary Analysis Platform for Cryptographic Protocols. In *ACM Conference on Computer and Communications Security (CCS)*.

[167] Matthias Neugschwandtner, Paolo Comparetti, Istvan Haller, and Herbert Bos. 2015. The BORG: Nanoprobing Binaries for Buffer Overreads. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*.

[168] Ivoline C Ngong, Brad Stenger, Joseph P Near, and Yuanyuan Feng. 2024. Evaluating the Usability of Differential Privacy Tools with Data Practitioners. In *Symposium on Usable Privacy and Security (SOUPS)*.

[169] Wode Ni, Joshua Sunshine, Vu Le, Sumit Gulwani, and Titus Barik. 2021. reCode: A Lightweight Find-and-Replace Interaction in the IDE for Transforming Code by Example. In *Annual ACM Symposium on User Interface Software and Technology (UIST)*.

[170] Olivier Nourry, Yutaro Kashiwa, Bin Lin, Gabriele Bavota, Michele Lanza, and Yasutaka Kamei. 2023. The Human Side of Fuzzing: Challenges Faced by Developers during Fuzzing Activities. *ACM Transactions on Software Engineering and Methodology* 33, 1, Article 14 (Nov. 2023), 26 pages. doi:10.1145/3611668

[171] Martin Nowack. 2019. Fine-Grain Memory Object Representation in Symbolic Execution. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*.

[172] Leysan Nurgalieva, Alisa Frik, and Gavin Doherty. 2023. A Narrative Review of Factors Affecting the Implementation of Privacy and Security Practices in Software Development. *Comput. Surveys* 55, 14s (2023), 1–27.

[173] Anderson Oliveira, João Correia, Wesley KG Assunção, Juliana Alves Pereira, Rafael de Mello, Daniel Coutinho, Caio Barbosa, Paulo Libório, and Alessandro Garcia. 2024. Understanding Developers' Discussions and Perceptions on Non-functional Requirements: The Case of the Spring Ecosystem. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 517–538.

[174] Stephen Oney and Joel Brandt. 2012. Codelets: Linking Interactive Documentation and Example Code in the Editor. In *ACM CHI Conference on Human Factors in Computing Systems (CHI)*.

[175] Ciprian Paduraru, Miruna Paduraru, and Alin Stefanescu. 2020. Optimizing Decision Making in Concolic Execution using Reinforcement Learning. In *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*.

[176] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. 2016. Shadow of a Doubt: Testing for Divergences between Software Versions. In *International Conference on Software Engineering (ICSE)*.

[177] Awanish Pandey, Phani Raj Goutham Kotcharlakota, and Subhajit Roy. 2019. Deferred Concretization in Symbolic Execution via Fuzzing. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.

[178] Darya Parygina, Alexey Vishnyakov, and Andrey Fedotov. 2022. Strong Optimistic Solving for Dynamic Symbolic Execution. In *Ivannikov Memorial Workshop (IVMEM)*.

[179] Corina S Păsăreanu and Willem Visser. 2009. A Survey of New Trends in Symbolic Execution for Software Testing and Analysis. *International Journal on Software Tools for Technology Transfer* 11, 4 (2009), 339–353.

[180] Nikhil Patnaik, Joseph Hallett, and Awais Rashid. 2019. Usability Smells: An Analysis of Developers' Struggle With Crypto Libraries. In *Symposium on Usable Privacy and Security (SOUPS)*.

[181] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: Fuzzing by Program Transformation. In *IEEE Symposium on Security and Privacy (S&P)*.

[182] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. 2020. VerX: Safety Verification of Smart Contracts. In *IEEE Symposium on Security and Privacy (S&P)*.

[183] Elizaveta Pertseva, Melinda Chang, Ulia Zaman, and Michael Coblenz. 2024. A Theory of Scientific Programming Efficacy. In *International Conference on Software Engineering (ICSE)*.

[184] Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, and Tevfik Bultan. 2017. Synthesis of Adaptive Side-Channel Attacks. In *IEEE Computer Security Foundations Symposium (CSF)*.

[185] Goran Piskachev, Matthias Becker, and Eric Bodden. 2023. Can the Configuration of Static Analyses Make Resolving Security Vulnerabilities More Effective? – A User Study. *Empirical Software Engineering* 28, 5 (2023), 118.

[186] 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 *Symposium on Usable Privacy and Security (SOUPS)*.

[187] Stephan Plöger, Mischa Meier, and Matthew Smith. 2023. A Usability Evaluation of AFL and libFuzzer with CS Students. In *ACM CHI Conference on Human Factors in Computing Systems (CHI)*.

[188] Sebastian Poeplau and Aurélien Francillon. 2019. Systematic Comparison of Symbolic Execution Systems: Intermediate Representation and its Generation. In *Annual Computer Security Applications Conference (ACSAC)*.

[189] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic Execution with SymCC: Don't Interpret, Compile!. In *USENIX Security Symposium*.

[190] Sebastian Poeplau and Aurélien Francillon. 2021. SymQEMU: Compilation-based Symbolic Execution for Binaries. In *Symposium on Network and Distributed System Security (NDSS)*.

[191] Minghui Quan. 2016. Hotspot Symbolic Execution of Floating-point Programs. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[192] Roshan N Rajapakse, Mansooreh Zahedi, M Ali Babar, and Haifeng Shen. 2022. Challenges and Solutions when Adopting DevSecOps: A Systematic Review. *Information and Software Technology* 141 (2022), 106700.

[193] David A Ramos and Dawson Engler. 2015. Under-Constrained Symbolic Execution: Correctness Checking for Real Code. In *USENIX Security Symposium*.

[194] Siddharth Prakash Rao, Gabriela Limonta, and Janne Lindqvist. 2022. Usability and Security of Trusted Platform Module (TPM) Library APIs. In *Symposium on Usable Privacy and Security (SOUPS)*.

[195] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*.

[196] Sebastian Roth, Lea Gröber, Michael Backes, Katharina Krombholz, and Ben Stock. 2021. 12 Angry Developers – A Qualitative Study on Developers' Struggles with CSP. In *ACM Conference on Computer and Communications Security (CCS)*.

[197] Sebastian Roth, Lea Gröber, Philipp Baus, Katharina Krombholz, and Ben Stock. 2024. Trust Me If You Can–How Usable is Trusted Types in Practice?. In *USENIX Security Symposium*.

[198] Reuben N. S. Rowe and James Brotherston. 2017. Automatic Cyclic Termination Proofs for Recursive Procedures in Separation Logic. In *ACM SIGPLAN Conference on Certified Programs and Proofs*.

[199] Nicola Ruaro, Fabio Gritti, Robert McLaughlin, Ilya Grishchenko, Christopher Kruegel, and Giovanni Vigna. 2024. Not your Type! Detecting Storage Collision Vulnerabilities in Ethereum Smart Contracts. In *Symposium on Network and Distributed System Security (NDSS)*.

[200] Nicola Ruaro, Fabio Pagani, Stefano Ortolani, Christopher Kruegel, and Giovanni Vigna. 2022. SYMBEXCEL: Automated Analysis and Understanding of Malicious Excel 4.0 Macros. In *IEEE Symposium on Security and Privacy (S&P)*.

[201] 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 *ACM International Conference Proceeding Series*.

[202] Marcel Ruoff, Brad A Myers, and Alexander Maedche. 2022. ONYX – User Interfaces for Assisting in Interactive Task Learning for Natural Language Interfaces of Data Visualization Tools. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts*.

[203] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* 61, 4 (2018), 58–66.

[204] Aakanksha Saha, James Mattei, Jorge Blasco, Lorenzo Cavallaro, Daniel Votipka, and Martina Lindorfer. 2025. Expert Insights into Advanced Persistent Threats: Analysis, Attribution, and Challenges. In *USENIX Security Symposium*.

[205] Charitha Saumya, Jinkyu Koo, Milind Kulkarni, and Saurabh Bagchi. 2019. XSTRESSOR: Automatic Generation of Large-Scale Worst-Case Test Inputs by Inferring Path Conditions. In *IEEE Conference on Software Testing, Validation and Verification (ICST)*.

[206] Daniel Schemmel, Julian Büning, Frank Busse, Martin Nowack, and Cristian Cadar. 2022. A Deterministic Memory Allocator for Dynamic Symbolic Execution. In *European Conference on Object-Oriented Programming (ECOOP)*.

[207] Daniel Schemmel, Julian Büning, César Rodríguez, David Laprell, and Klaus Wehrle. 2020. Symbolic Partial-Order Execution for Testing Multi-Threaded Programs. In *International Conference on Computer-Aided Verification (CAV)*. Springer International Publishing.

[208] Juliane Schmüser, Philip Klostermeyer, Kay Friedrich, and Sascha Fahl. 2025. "I'm pretty expert and I still screw it up": Qualitative Insights into Experiences and Challenges of Designing and Implementing Cryptographic Library APIs. In *IEEE Symposium on Security and Privacy (S&P)*.

[209] Eldon Schoop, Forrest Huang, and Bjoern Hartmann. 2021. Umlaut: Debugging Deep Learning Programs using Program Structure and Model Behavior. In *ACM CHI Conference on Human Factors in Computing Systems (CHI)*.

[210] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy (S&P)*.

[211] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005).

[212] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: Multi-path Symbolic Execution using Value Summaries. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[213] Tanusree Sharma, Zhixuan Zhou, Andrew Miller, and Yang Wang. 2023. A Mixed-Methods Study of Security Practices of Smart Contract Developers. In *USENIX Security Symposium*.

[214] Qingkai Shi, Junyang Shao, Yapeng Ye, Mingwei Zheng, and Xiangyu Zhang. 2023. Lifting Network Protocol Implementation to Precise Format Specification with Security Applications. In *ACM Conference on Computer and Communications Security (CCS)*.

[215] Shen Shiqi, Shweta Shinde, Soundarya Ramesh, Abhik Roychoudhury, and Prateek Saxena. 2019. Neuro-Symbolic Execution: Augmenting Symbolic Execution with Neural Constraints. In *Symposium on Network and Distributed System Security (NDSS)*.

[216] Yan Shoshitaishvili, Antonio Bianchi, Kevin Borgolte, Amat Cama, Jacopo Corbetta, Francesco Disperati, Audrey Dutcher, John Grosen, Paul Grosen, Aravind Machiry, et al. 2018. Mechanical Phish: Resilient Autonomous Hacking. *IEEE Security & Privacy* 16, 2 (2018), 12–22.

[217] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Symposium on Network and Distributed System Security (NDSS)*.

[218] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok: (State of) the Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy (S&P)*.

[219] Ziqi Shuai, Zhenbang Chen, Yufeng Zhang, Jun Sun, and Ji Wang. 2021. Type and Interval Aware Array Constraint Solving for Symbolic Execution. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*.

[220] Jonathan Sillito, Gail C Murphy, and Kris De Volder. 2006. Questions Programmers Ask during Software Evolution Tasks. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*.

[221] Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bill Chu, and Heather Richter Lipford. 2015. Questions Developers Ask while Diagnosing Potential Security Vulnerabilities with Static Analysis. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[222] Justin Smith, Christopher Theisen, and Titus Barik. 2020. A Case Study of Software Security Red Teams at Microsoft. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*.

[223] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. 2021. SMARTEST: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution. In *USENIX Security Symposium*.

[224] Benjamin Steenhoek, Kalpathy Sivaraman, Renata Saldivar Gonzalez, Yevhen Mohylevskyy, Roshanak Zilouchian Moghaddam, and Wei Le. 2024. Closing the Gap: A User Study on the Real-world Usefulness of AI-powered Vulnerability Detection & Repair in the IDE. *arXiv preprint arXiv:2412.14306* (2024).

[225] 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 *Symposium on Network and Distributed System Security (NDSS)*.

[226] Anselm Strauss and Juliet Corbin. 1990. *Basics of Qualitative Research*. Vol. 15. Newbury Park, CA: Sage.

[227] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. 2018. Concolic Testing for Deep Neural Networks. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*.

[228] Chungha Sung, Brandon Paulsen, and Chao Wang. 2018. CANAL: A Cache Timing Analysis Framework via LLVM Transformation. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*.

[229] Mohammad Tahaei and Kami Vaniea. 2019. A Survey on Developer-Centred Security. In *IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*.

[230] Mohammad Tahaei, Kami Vaniea, Konstantin Beznosov, and Maria K Wolters. 2021. Security Notifications in Static Analysis Tools: Developers' Attitudes, Comprehension, and Ability to Act on Them. In *ACM CHI Conference on Human Factors in Computing Systems (CHI)*.

[231] Ningzhi Tang, Meng Chen, Zheng Ning, Aakash Bansal, Yu Huang, Collin McMillan, and T Li. 2023. An Empirical Study of Developer Behaviors for Validating and Repairing AI-generated Code. In *Workshop on the Intersection of HCI and PL*.

[232] Tyler Thomas, Bill Chu, Heather Lipford, Justin Smith, and Emerson Murphy-Hill. 2015. A Study of Interactive Code Annotation for Access Control Vulnerabilities. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*.

[233] Tyler W Thomas, Heather Lipford, Bill Chu, Justin Smith, and Emerson Murphy-Hill. 2016. What Questions Remain? An Examination of how Developers Understand an Interactive Static Analysis Tool. In *Symposium on Usable Privacy and Security (SOUPS)*.

[234] Ronald E. Thompson, Madline McLaughlin, Carson Powers, and Daniel Votipka. 2024. "There are rabbit holes I want to go down that I'm not allowed to go down": An Investigation of Security Expert Threat Modeling Practices for Medical Devices. In *USENIX Security Symposium*.

[235] David Trabish, Shachar Itzhaky, and Noam Rinetzky. 2021. A Bounded Symbolic-size Model for Symbolic Execution. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[236] David Trabish, Timotej Kapus, Noam Rinetzky, and Cristian Cadar. 2020. Past-sensitive Pointer Analysis for Symbolic Execution. In *ACM Joint European*

Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).
[237] David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. 2018. Chopped Symbolic Execution. In *International Conference on Software Engineering (ICSE).*
[238] David Trabish and Noam Rinetzky. 2020. Relocatable Addressing Model for Symbolic Execution. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).*
[239] Asif Kamal Turzo and Amiangshu Bosu. 2024. What Makes a Code Review Useful to Opendev Developers? An Empirical Investigation. *Empirical Software Engineering* 29, 1 (2024), 6.
[240] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Andy Zaidman, and Harald C Gall. 2018. Context is King: The Developer Perspective on the Usage of Static Analysis Tools. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER).*
[241] Alexey Vishnyakov, Andrey Fedotov, Daniil Kuts, Alexander Novikov, Darya Parygina, Eli Kobrin, Vlada Logunova, Pavel Belecky, and Shamil Kurmangaleev. 2020. Sydr: Cutting Edge Dynamic Symbolic Execution. In *Ivannikov Ispras Open Conference (ISPRAS).* IEEE.
[242] Alexey Vishnyakov, Daniil Kuts, Vlada Logunova, Darya Parygina, Eli Kobrin, Georgy Savidov, and Andrey Fedotov. 2022. Sydr-Fuzz: Continuous Hybrid Fuzzing and Dynamic Analysis for Security Development Lifecycle. In *Ivannikov Ispras Open Conference (ISPRAS).*
[243] Alexey Vishnyakov, Vlada Logunova, Eli Kobrin, Daniil Kuts, Darya Parygina, and Andrey Fedotov. 2021. Symbolic Security Predicates: Hunt Program Weaknesses. In *Ivannikov Ispras Open Conference (ISPRAS).*
[244] 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 *USENIX Security Symposium.*
[245] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S. Foster, and Michelle L. Mazurek. 2020. An Observational Investigation of Reverse Engineers' Processes. In *USENIX Security Symposium.*
[246] Daniel Votipka, Rock Stevens, Elissa Redmiles, Jeremy Hu, and Michelle Mazurek. 2018. Hackers vs. Testers: A Comparison of Software Vulnerability Discovery Processes. In *IEEE Symposium on Security and Privacy (S&P).*
[247] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. 2020. KLEESpectre: Detecting Information Leakage through Speculative Cache Attacks via Symbolic Execution. *ACM Transactions on Software Engineering and Methodology* 29, 3 (2020), 1–31.
[248] Yuanpeng Wang, Ziqi Zhang, Ningyu He, Zhineng Zhong, Shengjian Guo, Qinkun Bao, Ding Li, Yao Guo, and Xiangqun Chen. 2023. SymGX: Detecting Cross-boundary Pointer Vulnerabilities of SGX Applications via Static Symbolic Execution. In *ACM Conference on Computer and Communications Security (CCS).*
[249] Zhongjie Wang, Shitong Zhu, Yue Cao, Zhiyun Qian, Chengyu Song, Srikanth V. Krishnamurthy, Kevin S. Chan, and Tracy D. Braun. 2020. SymTCP: Eluding Stateful Deep Packet Inspection with Automated Discrepancy Discovery. In *Symposium on Network and Distributed System Security (NDSS).*
[250] Guannan Wei, Songlin Jia, Ruiqi Gao, Haotian Deng, Shangyin Tan, Oliver Bračevac, and Tiark Rompf. 2023. Compiling Parallel Symbolic Execution with Continuations. In *International Conference on Software Engineering (ICSE).*
[251] Charles Weir, Awais Rashid, and James Noble. 2016. How to Improve the Security Skills of Mobile App Developers? Comparing and Contrasting Expert Views. In *Symposium on Usable Privacy and Security (SOUPS).*
[252] Hongbo Wen, Hanzhi Liu, Jiaxin Song, Yanju Chen, Wenbo Guo, and Yu Feng. 2024. FORAY: Towards Effective Attack Synthesis against Deep Logical Vulnerabilities in DeFi Protocols. In *ACM Conference on Computer and Communications Security (CCS).*
[253] Junye Wen, Tarek Mahmud, Meiru Che, Yan Yan, and Guowei Yang. 2023. Intelligent Constraint Classification for Symbolic Execution. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER).*
[254] Dominik Wermke, Noah Wöhler, Jan H Klemmer, Marcel Fourné, Yasemin Acar, and Sascha Fahl. 2022. Committed to Trust: A Qualitative Study on Security & Trust in Open Source Software Projects. In *IEEE Symposium on Security and Privacy (S&P).*
[255] Jim Witschey, Shundan Xiao, and Emerson Murphy-Hill. 2014. Technical and Personal Factors Influencing Developers' Adoption of Security Tools. In *ACM Workshop on Security Information Workers.*
[256] Edmund Wong, Lei Zhang, Song Wang, Taiyue Liu, and Lin Tan. 2015. DASE: Document-Assisted Symbolic Execution for Improving Automated Software Testing. In *International Conference on Software Engineering (ICSE).*
[257] Miuyin Yong Wong, Matthew Landen, Frank Li, Fabian Monrose, and Mustaque Ahamad. 2024. Comparing Malware Evasion Theory with Practice: Results from Interviews with Expert Analysts. In *Symposium on Usable Privacy and Security (SOUPS).*
[258] Novia Wong, Nai-Yu Cheng, Bruna Oewel, Katherine E Genuario, SarahElizabeth Stoeckl, Stephen M Schueller, Iftekhar Ahmed, André van der Hoek, and Madhu Reddy. 2025. 'It's a spectrum': Exploring Autonomy, Competence, and Relatedness in Software Development Processes and Tools. In *ACM CHI Conference on Human Factors in Computing Systems (CHI).*
[259] Hongwei Wu, Jianliang Wu, Ruoyu Wu, Ayushi Sharma, Aravind Machiry, and Antonio Bianchi. 2025. VeriBin: Adaptive Verification of Patches at the Binary Level. In *Symposium on Network and Distributed System Security (NDSS).*
[260] Mingyuan Wu, Jiahong Xiang, Kunqiu Chen, Peng Di, Shin Hwei Tan, Heming Cui, and Yuqun Zhang. 2024. Tumbling Down the Rabbit Hole: How do Assisting Exploration Strategies Facilitate Grey-box Fuzzing?. In *International Conference on Software Engineering (ICSE).*
[261] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. 2018. FUZE: Towards Facilitating Exploit Generation for Kernel Use-After-Free Vulnerabilities. In *USENIX Security Symposium.*
[262] Haoyu Xiao, Yuan Zhang, Minghang Shen, Chaoyang Lin, Can Zhang, Shengli Liu, and Min Yang. 2024. Accurate and Efficient Recurring Vulnerability Detection for IoT Firmware. In *ACM Conference on Computer and Communications Security (CCS).*
[263] Jing Xie, Bill Chu, Heather Richter Lipford, and John Melton. 2011. ASIDE: IDE Support for Web Application Security. In *Annual Computer Security Applications Conference (ACSAC).*
[264] Jing Xie, Heather Richter Lipford, and Bill Chu. 2011. Why Do Programmers Make Security Errors?. In *IEEE Symposium on Visual Languages and Human Centric Computing (VL/HCC).*
[265] XMUsuny. 2025. Symbolic Execution Papers. https://github.com/XMUsuny/symbolic-execution-papers Accessed: 2025-04-08.
[266] Hui Xu, Zirui Zhao, Yangfan Zhou, and Michael R. Lyu. 2020. Benchmarking the Capability of Symbolic Execution Tools with Logic Bombs. *IEEE Transactions on Dependable and Secure Computing* 17, 6 (2020).
[267] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. 2018. Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels. In *IEEE Symposium on Security and Privacy (S&P).*
[268] Babak Yadegari and Saumya Debray. 2015. Symbolic Execution of Obfuscated Code. In *ACM Conference on Computer and Communications Security (CCS).*
[269] Carter Yagemann, Simon P Chung, Brendan Saltaformaggio, and Wenke Lee. 2021. Automated Bug Hunting with Data-driven Symbolic Root Cause Analysis. In *ACM Conference on Computer and Communications Security (CCS).*
[270] 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 *USENIX Security Symposium.*
[271] Rei Yamagishi, Shota Fujii, Shingo Yasuda, Takayuki Sato, and Ayako A. Hasegawa. 2025. Collaborative Work in Malware Analysis: Understanding the Roles and Challenges of Malware Analysts. In *ACM CHI Conference on Human Factors in Computing Systems (CHI).*
[272] Guowei Yang, Rui Qiu, Sarfraz Khurshid, Corina S. Păsăreanu, and Junye Wen. 2019. A Synergistic Approach to Improving Symbolic Execution using Test Ranges. *Innovations in Systems and Software Engineering* 15, 3 (2019).
[273] Mingxuan Yao, Runze Zhang, Haichuan Xu, Shih-Huan Chou, Varun Chowdhary Paturi, Amit Kumar Sikder, and Brendan Saltaformaggio. 2024. Pulling off the Mask: Forensic Analysis of the Deceptive Creator Wallets behind Smart Contract Fraud. In *IEEE Symposium on Security and Privacy (S&P).*
[274] Peisen Yao, Qingkai Shi, Heqing Huang, and Charles Zhang. 2020. Fast Bit-vector Satisfiability. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).*
[275] Tuba Yavuz. 2022. SIFT: A Tool for Property Directed Symbolic Execution of Multithreaded Software. In *IEEE Conference on Software Testing, Verification and Validation (ICST).*
[276] Qiuping Yi and Jeff Huang. 2018. Concurrency Verification with Maximal Path Causality. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).*
[277] Qiuping Yi, Zijiang Yang, Shengjian Guo, Chao Wang, Jian Liu, and Chen Zhao. 2018. Eliminating Path Redundancy via Postconditioned Symbolic Execution. *IEEE Transactions on Software Engineering* 44, 1 (2018).
[278] Miuyin Yong Wong, Matthew Landen, Manos Antonakakis, Douglas M. Blough, Elissa M. Redmiles, and Mustaque Ahamad. 2021. An Inside Look into the Practice of Malware Analysis. In *ACM Conference on Computer and Communications Security (CCS).*
[279] Hengbiao Yu, Zhenbang Chen, Ji Wang, Zhendong Su, and Wei Dong. 2018. Symbolic Verification of Regular Properties. In *International Conference on Software Engineering (ICSE).*
[280] Hengbiao Yu, Zhenbang Chen, Yufeng Zhang, Ji Wang, and Wei Dong. 2017. RGSE: A Regular Property Guided Symbolic Executor for Java. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).*
[281] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *USENIX Security Symposium.*
[282] Yicheng Zeng, Jiaqian Peng, Zhihui Zhao, Zhanwei Song, Hongsong Zhu, and Limin Sun. 2022. SIFOL: Solving Implicit Flows in Loops for Concolic Execution.

In *IEEE International Performance, Computing, and Communications Conference (IPCCC)*.
[283] Wuqi Zhang, Zhuo Zhang, Qingkai Shi, Lu Liu, Lili Wei, Yepang Liu, Xiangyu Zhang, and Shing-Chi Cheung. 2024. Nyx: Detecting Exploitable Front-running Vulnerabilities in Smart Contracts. In *IEEE Symposium on Security and Privacy (S&P)*.
[284] Yufeng Zhang, Zhenbang Chen, Ziqi Shuai, Tianqi Zhang, Kenli Li, and Ji Wang. 2020. Multiplex Symbolic Execution: Exploring Multiple Paths by Solving Once. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*.
[285] Yufeng Zhang, Zhenbang Chen, Ji Wang, Wei Dong, and Zhiming Liu. 2015. Regular Property Guided Dynamic Symbolic Execution. In *International Conference on Software Engineering (ICSE)*.
[286] Yuexi Zhang, Bingyu Li, Jingqiang Lin, Linghui Li, Jiaju Bai, Shijie Jia, and Qianhong Wu. 2024. Gopher: High-Precision and Deep-Dive Detection of Cryptographic API Misuse in the Go Ecosystem. In *ACM Conference on Computer and Communications Security (CCS)*.
[287] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. 2019. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*.
[288] Yunze Zhao, Wentao Guo, Harrison Goldestin, Daniel Votipka, Kelsey R. Fulton, and Michelle L. Mazurek. 2025. A Qualitative Analysis of Fuzzing Tool Usability and Challenges. In *ACM Conference on Computer and Communications Security (CCS)*.
[289] 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 *USENIX Security Symposium*.
[290] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. 2021. Automatic Firmware Emulation through Invalidity-guided Knowledge Inference. In *USENIX Security Symposium*.
[291] Jun Zhu, Heather Richter Lipford, and Bill Chu. 2013. Interactive Support for Secure Programming Education. In *ACM Technical Symposium on Computer Science Education (SIGCSE)*.
[292] Conrad Zimmerman, Jenna DiVincenzo, and Jonathan Aldrich. 2024. Sound Gradual Verification with Symbolic Execution. *ACM on Programming Languages* 8, POPL (2024), 85:2547–85:2576.
[293] Xiaochen Zou, Guoren Li, Weiteng Chen, Hang Zhang, and Zhiyun Qian. 2022. SyzScope: Revealing High-Risk Security Impacts of Fuzzer-Exposed Bugs in Linux Kernel. In *USENIX Security Symposium*.

A SE Literature Codebooks

In this section, we give the final codebooks we used for the analysis. Each table describes a type of code with all the relevant sub-code types defined. We also report the final count of each sub-code type and the final α values reached when calculating the agreement.

Received 2026-02-02; accepted 2026-03-19

SE Implementation	**Definition**	**Count**	α
Level of SE			N/A
Intermediate Language	When the SE engine operates on intermediate representation language like LLVM	103	
Binary	When the SE engine operates directly on the binary file	27	
Source Code	When the SE engine operates directly on source code	27	
Base Engine / Solver			N/A
Z3	When the SE implementation uses Z3 as a component	10	
S2E	When the SE implementation uses S2E as a component	4	
Triton	When the SE implementation uses Triton as a component	3	
CREST	When the SE implementation uses CREST as a component	5	
QSYM	When the SE implementation uses QSYM as a component	3	
JPF	When the SE implementation uses JPF as a component	4	
Angr	When the SE implementation uses Angr as a component	24	
KLEE	When the SE implementation uses KLEE as a component	58	

Table 8: Codes related to how SE was implemented. Table of what language representation SE operates on and what base engine or solver was utilized. Base engines with two or fewer uses are omitted for space.

Search Heuristic	**Definition**	**Count**	α
Path Operation			.86
Path Selection	When the *Search Heuristic* focuses on choosing the next path to explore	37	
Path Consolidation	When the *Search Heuristic* consolidates similar paths together	9	
Path Removal	When the *Search Heuristic* prunes paths from the search space	41	
Operation Metrics			1
Execution Context	When the *Search Heuristic* is informed by dynamic runtime information	14	
Conventional Heuristic	When the *Search Heuristic* uses a novel algorithm like BFS/DFS	7	
Symbolic Values	When the *Search Heuristic* is informed by SSI such as σ values	24	
Program Semantics	When the *Search Heuristic* utilizes program structure or control flow information	30	
Coverage	When the *Search Heuristic* utilizes code coverage to inform decisions	17	
Target	When the *Search Heuristic* is selecting paths based on a predetermined location to reach in the code	12	

Table 9: Codes related to how the *Search Heuristic* was improved. Table of how the *Search Heuristic* manages different potential execution paths and the metrics that inform its decisions.

SE module	**Definition**	**Count**	α
Execution Driver			1
Binary Instrumentation	When the target binary file is instrumented with SE optimizations before execution	7	
Parallelization & Forking	When improvements to the *Execution Driver* involve symbolically executing in parallel or forking	16	
Selective SE	When the *Execution Driver* is either selectively invoked or has a specified endpoint	17	

Table 10: Codes related to how the *Execution Driver* was improved. Table of how the *Execution Driver* was improved through selective invocation or runtime optimizations.

SE module	Definition	Count	α
Environment and Memory Model			1
Memory values	When the *Environment and Memory Model* improvement is specifically for memory values	18	
Function model	When the *Environment and Memory Model* improvement is specifically for modeling functions	15	
Firmware or Input/Output	When the *Environment and Memory Model* improvement is specifically for modeling firmware or memory-mapped inputs and outputs	7	

Table 11: Codes related to how the *Environment and Memory Model* was improved. Table of how the *Environment and Memory Model* was improved, either modeling functions, memory values, or firmware.

SE module	Definition	Count	α
Symbolic State Manager			1
Memory Allocator	When the *Symbolic State Manager* improvement is based on how symbolic states are allocated, such as new policies for memory allocation.	11	
Memory Representation	When the *Symbolic State Manager* improvement is based on the structure of the stored symbolic states, like defining custom tree objects.	24	

Table 12: Codes related to how the *Symbolic State Manager* was improved. Table of how the *Symbolic State Manager* was improved, either improved memory allocation policies or data structures for storing SSI.

SE module	Definition	Count	α
Constraint Solver			.89
Constraint Representation	When the π constraints are represented in a novel way to boost solver performance	17	
Constraint Simplification	When the π constraints provided to the *Constraint Solver* are simplified to improve performance.	14	
Selective Invocation	When the *Constraint Solver* is selectively invoked to reduce time spent solving constraints	9	
Caching Results	When the results from prior calls to the *Constraint Solver* are shared for future calls to boost performance	11	

Table 13: Codes related to how the *Constraint Solver* was improved. Table of how the *Constraint Solver* was improved, either changing how π constraints are represented, simplified, reused, or by selectively invoking the *Constraint Solver*.

Code Type	**Definition**	**Count**	α
Mentioned SE Limitation			N/A
Constraint Solving Cost	When the paper mentions constraint solving cost as a bottleneck for SE	51	
Memory Cost	When the paper mentions the high memory cost of storing SSI as a bottleneck for SE	17	
Accuracy	When the paper mentions potential false positives, false negatives or other accuracy measures as an issue for SE	20	
Memory Modeling	When the paper mentions difficulties with modeling memory values during SE	16	
Environment Modeling	When the paper mentions difficulties with modeling other aspects of the environment including functions, system calls, or external calls during SE	34	
Path Explosion	When the paper mentions path explosions or describes the ramping scale of potential paths.	92	
Code Coverage	When the paper mentions issues with achieving high code coverage during SE	20	

Table 14: Codes related to how SE limitations. Table of how the different limitations mentioned in the papers in our corpus. This includes path explosion, high computational and memory overheads, and accuracy.

Code Type	**Definition**	**Count**	α
Improved SE Limitation			.8
Constraint Solving Cost	When the paper makes an improvement affecting constraint solving cost as a bottleneck for SE	31	
Memory Cost	When the paper makes an improvement affecting the memory cost of storing SSI as a bottleneck for SE	7	
Accuracy	When the paper makes an improvement affecting the overall accuracy for SE	7	
Memory Modeling	When the paper improves modeling memory values during SE	9	
Environment Modeling	When the paper improves modeling other aspects of the environment, including functions, system calls, or external calls during SE	16	
Path Explosion	When the paper improves the path explosion problem by limiting the search space.	47	
Code Coverage	When the paper improves code coverage during SE	37	

Table 15: Codes related to improved SE limitations. Table of the different limitations improved by a paper's contributions.

Code Type	**Definition**	**Count**	α
SE Use Case			.81
Defensive Security	When SE is used for alternative defensive security tasks like honeypots, or evasion of detection.	6	
Malware Analysis	When SE is used to help analyze malware samples.	8	
Taint Analysis	When SE is used to track data flows or perform taint analysis.	6	
Hardware/Firmware Analysis	When SE is used to analyze firmware or hardware/IoT devices.	14	
Side-channel detection	When SE is used to detect side-channel vulnerabilities in programs.	10	
Bug Discovery	When SE is used to discover memory, logical, or other arithmetic bugs in programs.	102	
Verification	When SE is used to verify program correctness, or other properties, including security.	45	
Test Case Generation	When SE is used specifically for its ability to generate test cases for programs.	73	

Table 16: Codes related to SE use cases. Table of the different use cases of SE that the papers leveraged.

Code Type	**Definition**	**Count**	α
Evaluation Metrics			.86
F1 Score / FP/FN	When SE accuracy is evaluated with an F1 score or the number of false positives and false negatives.	10	
Number of States	When the number of created symbolic execution paths or states measures SE efficiency.	11	
Number of *Constraint Solver* Calls	When the number of times the *Constraint Solver* is invoked measures SE efficiency.	12	
Number of Bugs	When the number of bugs is used to evaluate SE effectiveness.	63	
Memory Usage	When the amount of memory consumption is used to evaluate SE efficiency.	3	
Time Run	When the amount of time SE is run is used to evaluate its effectiveness and efficiency.	87	
Code Coverage	When the code coverage percentage is used to evaluate SE effectiveness.	79	
Constraint Solver Time	When the amount of time the *Constraint Solver* is run is used to measure SE efficiency.	6	

Table 17: Codes related to SE evaluation methods. Table of the different metrics used by the papers in our corpus to evaluate their technical improvements to SE.

Code Type	**Definition**	**Count**	α
Method Limitations			.86
Code Coverage	When the proposed SE improvement suffers from decreased code coverage in some circumstances.	4	
Accuracy	When the proposed SE improvement suffers from decreased accuracy with identified bugs.	18	
Overhead	When the proposed SE improvement suffers from increased computation or memory overhead.	7	
Scale	When the proposed SE improvement suffers from scalability issues when applied to larger programs.	10	
Domain Restrictions	When the proposed SE improvement is limited in where it can be applied, either by language or type of application.	34	

Table 18: Codes related to SE method limitations. Table of the different limitations mentioned by the authors specific to their technical improvements to SE.