A Qualitative Analysis of Fuzzer Usability and Challenges

Yunze Zhao
University of Maryland
College Park, MD, USA
yunze@umd.edu

Wentao Guo
University of Maryland
College Park, MD, USA
wguo5@umd.edu

Harrison Goldstein
University of Maryland
College Park, MD, USA
harrygol@umd.edu

Daniel Votipka
Tufts University
Medford, MA, USA
dvotipka@cs.tufts.edu

Kelsey R. Fulton
Colorado School of Mines
Golden, CO, USA
kelsey.fulton@mines.edu

Michelle L. Mazurek
University of Maryland
College Park, MD, USA
mmazurek@umd.edu

Abstract

Fuzzing is a widely adopted technique for uncovering software vulnerabilities by generating random or mutated test inputs to trigger unexpected behavior. However, little is known about how developers actually use fuzzing tools in practice, the challenges they face, and where current tools fall short. This study investigates the human side of fuzzing via 18 semi-structured interviews with fuzzing users across diverse domains. These interviews explore participants' workflows, frustrations, and expectations around fuzzing, revealing critical usability gaps and design opportunities. Our results can inform the next generation of fuzzing tools to improve user experience, reduce manual effort, and enable more effective integration of fuzzing into real-world workflows.

CCS Concepts

• **Security and privacy → Usability in security and privacy**.

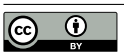
Keywords

Fuzzing, Usable Security, Usability, Dynamic Testing

ACM Reference Format:
Yunze Zhao, Wentao Guo, Harrison Goldstein, Daniel Votipka, Kelsey R. Fulton, and Michelle L. Mazurek. 2025. A Qualitative Analysis of Fuzzer Usability and Challenges. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3765055

1 Introduction

Automated techniques for identifying security vulnerabilities have become essential tools for developers and security professionals. Among these techniques, fuzzing has emerged as a widely adopted automated testing method, crucial for discovering software vulnerabilities by generating random inputs and evaluating how programs handle them. Its ability to uncover unexpected behaviors and critical security flaws has made it indispensable in open-source and commercial software development projects [13, 42, 52, 67, 81].



CCS '25, Taipei, Taiwan

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3765055

The efficacy of fuzzing in vulnerability discovery is well documented, with tools (*fuzzers*) such as AFL/AFL++, FuzzTest, and OSS-Fuzz leading the way in identifying thousands of vulnerabilities across widely used software, including complex systems such as browsers and kernels [5, 12, 14, 17, 48]. Consequently, fuzzing is increasingly recognized as a critical and powerful concept in modern software testing pipelines and is recommended in various industry standards and security guidelines, underscoring its growing role in secure software development [4, 25, 60].

The majority of fuzzing research has prioritized technical advancements, such as input generation [34, 68], seed scheduling [56, 80], mutation [7, 28], and harness generation [2, 21, 57]. However, this focus on technical improvement has largely overlooked how users interact with and adapt these tools in practice, leading to usability challenges that are not mere inconveniences but significant barriers that directly undermine the security benefits these tools aim to provide.

Initial usability research—primarily observational or experimental studies with students and/or in lab settings—has begun to identify some potentially important challenges. For example, fuzzers tend to exhibit a steep learning curve, reducing their accessiblity to non-expert developers [3, 45, 49]. Grey-box fuzzers like AFL and libFuzzer rely on code coverage feedback to guide testing and discover bugs, but configuring and tuning these tools for optimal performance demand a deep understanding of the underlying mechanisms [5, 14, 81]. Challenges with configuration and associated workflows can lead to frustration and misconfiguration, reducing the usage of fuzzing in everyday software development practices [31, 81]. Further, a lack of standardized evaluation practices makes it difficult for users to compare options based on performance [19, 27]

These prior studies have made an important start toward understanding fuzzer usability, but they have limited visibility into how experienced practitioners fit fuzzers into their real-world development and vulnerability analysis workflows more broadly.

This paper attempts to address this gap through 18 semi-structured interviews with experienced users of fuzzers from both academia and industry. Semi-structured interviews provide the flexibility to qualitatively explore the specific struggles and obstacles faced by practitioners, while also capturing a broad range of insights into their workflows, challenges, and suggestions for improvement.

Specifically, we consider the following key research questions:

RQ1: What specific challenges do users face across the lifecycle of a fuzzing campaign in real-world deployments?

RQ2: What strategies do practitioners use to address or work around these challenges?
RQ3: What improvements to fuzzers would better support practical adoption and real-world workflows?

Our findings reveal that while participants view fuzzing as powerful and indispensable, they frequently encounter steep learning curves, unintuitive interfaces, and limited feedback during both setup and analysis. Participants often rely on informal heuristics and self-directed learning, leading to incomplete or inconsistent mental models of how fuzzers operate. Many reported that fuzzers are difficult to configure and rarely integrate well with modern development workflows. Despite these barriers, participants demonstrated creative adaptations to make fuzzers useful and expressed a strong desire for tools that offer more actionable guidance, better integration with existing workflows, and smarter, more transparent interfaces.

2 Background and related work

In this section, we provide a brief introduction to fuzzing, including fundamental concepts, popular techniques, and widely used tools. Then we discuss studies on the usability of fuzzers and of security tools more broadly.

2.1 Fundamental concepts of fuzzing

Fuzzing is a form of dynamic software testing that evaluates program behavior by executing code with a wide range of inputs, then observing runtime outcomes. A fundamental assumption in fuzzing is that reliability flaws and security flaws will manifest as detectable failures when exposed to sufficiently diverse inputs. At its core, a **fuzzing campaign** operates by generating inputs to the **fuzzing target** (the application, library, or system under test), executing it, and monitoring for failures such as crashes, hangs, or error signals [42]. Fuzzers sometimes begin with a set of user-provided inputs (**seeds**) and progressively mutate them through a **mutator** component to create new test cases. Inputs that explore new paths or trigger interesting behavior are used to guide further fuzzing target exploration. To connect the fuzzer to program logic, users typically write a **harness**: a wrapper function that accepts fuzzer inputs and passes them to fuzzing target.

Fuzzers are often classified by how much visibility they have into the target's internals. Black-box fuzzers treat the program as opaque, relying solely on observable outputs like crashes. Conversely, white-box fuzzers leverage program structure through techniques like symbolic execution. In between, grey-box fuzzers combine lightweight instrumentation with runtime feedback to guide input generation. The most common form of runtime feedback is **coverage**: a measure of how much of the program (e.g., branches, paths, or functions) has been exercised [16, 30, 42, 73, 74, 81]. Coverage-guided grey-box fuzzers (e.g., AFL, AFL++, libFuzzer, and Honggfuzz) dominate both research and practice due to their balance between effectiveness and scalability [14, 81].

Our participants primarily reported experience with these widely used tools; as such, coverage-guided grey-box fuzzing is the main focus of our study.

2.2 Workflow of coverage-guided fuzzing

While the specifics of fuzzing vary depending on the use case [30, 42, 73, 74, 81], the typical workflow of coverage-guided fuzzers requires users to actively prepare the target, integrate the fuzzer, and interpret dynamic behavior and results.

Building the target with fuzzing instrumentation. First, the target application must be instrumented to enable runtime feedback [70]. This typically requires using custom compilers or compiler flags. This step may also require linking against specific runtime libraries or enabling sanitizers for memory error detection [32].

Connecting fuzzing inputs to the target. Users must next ensure that generated fuzzing inputs are routed into the program logic they want to test. This can take different forms depending on the fuzzer and the target application's structure.

Some fuzzers, such as libFuzzer [33], expect users to supply a harness written in the fuzz target's source code and compiled together with the fuzzer. Creating a harness requires understanding both the application's API and how the fuzzer invokes the harness.

In contrast, tools like AFL and AFL++ typically fuzz full programs through the command-line interface [14]. These tools inject inputs via stdin or temporary files, and they monitor for crashes or unusual behavior. This approach is simpler because it does not require modifying source code, but it offers less control over targeting specific code paths or functions.

Most fuzzers generate unstructured, byte-level inputs, which can be useful for raw parsing logic but require further configuration by users for complex, structured input formats.

Recent research has explored automating input delivery, usually through harness generation systems [2, 22, 77, 79]. More recently, large language models have been employed to automatically generate harnesses [10, 37, 71, 76, 78], showing promise in reducing manual setup effort. However, these automated approaches, while powerful, often still require manual refinement for complex APIs or targets.

Configuring input strategy and seeds. Once input delivery is established, users sometimes need to configure how the fuzzer generates inputs to explore different execution paths [30]. Most modern fuzzers come with optimized, built-in mutation strategies that can be effective out of the box [20, 28, 35, 36, 53]. Instead of adjusting the mutation strategy, users typically provide example seed inputs, enable grammar support [69], or select from advanced options. In particular, selecting high-quality initial seeds can accelerate path exploration, improve code coverage, and reduce the number of iterations needed to discover bugs [20, 53].

Monitoring and interpreting program behavior. In coverage-guided fuzzing, users track dynamic feedback such as code coverage, execution counts, or branch exploration to assess whether the fuzzer is exercising new paths. They can then adjust the configuration accordingly.

While running, fuzzers monitor for anomalous behavior such as crashes, memory violations, assertion failures, or timeouts by using lightweight instrumentation or integrations with external

sanitizers. They report on these anomalies, providing output such as crash logs and stack or execution traces. Users must interpret this output to understand the (possible) bugs the fuzzer has identified.

2.3 Fuzzer usability

Next, we turn to existing research examining fuzzer usability, which can be divided into two categories. First, some work has compared various tools and usage contexts in controlled experiments [44, 49, 50]. For example, Nosco et al. conducted an experiment with 12 security professionals to compare the efficacy of breadth- vs. depth-first approaches to fuzzing target selection [44]. In the process, they identified challenges faced when participants attempted to set up the fuzzers.

Plöger et al. investigated usability more directly, examining how CTF participants and computer-science students engage with common fuzzers. In a first study comparing libFuzzer to the Clang static analyzer, most participants struggled to set up the fuzzing target [49]. In a follow-up comparing libFuzzer to AFL, the authors found that even with clear goals and scaffolding to assist with setup, student participants still struggled with both tools during instrumentation, result interpretation, and harness creation [50].

While these studies provide valuable guidance toward improving fuzzer usability, the restricted setting and generally limited fuzzing experience of participants limits the scope of possible findings. For example, participants' relative inexperience often prevented them from moving beyond initial setup, confining the identified challenges to the early stages of the fuzzing process. In contrast, our study focuses on practitioners with long-term experience fuzzing real-world systems, enabling us to identify a broader range of challenges, as well as strategies employed by practitioners to overcome these challenges. By conducting semi-structured interviews, we are able to explore in detail the nuanced contexts, mental models, and workarounds that define modern fuzzing practice.

The second category of related work identifies fuzzing challenges broadly through surveys, expert discussions, and artifact analyses [6, 45]. Böhme et al. synthesized discussions from a working group of 31 fuzzing practitioners and researchers, identifying multiple fuzzing challenges [6]. They then surveyed 21 leaders in industry and academia outside the working group to confirm these challenges. They identify usability as one concern, emphasizing that while fuzzing simplifies bug discovery, it still requires significant expertise; however, they do not explore concrete usability challenges and their mitigations in depth.

Similarly, Nourry et al. reviewed GitHub issues associated with OSS-Fuzz to produce a taxonomy of reported fuzzing challenges, then validated by surveying 103 developers about their experiences with each challenge [45]. Their taxonomy includes usability challenges, such as selecting appropriate fuzzers and configuring targets; however, it is limited to challenges that appear in GitHub issues, and it does not detail how these challenges manifest in practice or how fuzzing users overcome them.

Both studies take valuable steps in identifying the importance of usability for fuzzing in practice. Our study expands on this work by capturing a broader range of perspectives, using semi-structured interviews to explore usability challenges (and their potential solutions) in detail.

2.4 Usability beyond fuzzing

Next, we briefly survey the growing body of research investigating and attempting to improve the usability of other (non-fuzzing) security tools.

Static application security testing (SAST) tools, which analyze source code, bytecode, or binaries without executing the program, are particularly well studied. Prior work has shown these tools may produce false positives that can overwhelm users [11, 18, 29, 58, 63, 64, 66], lack clear outputs [11, 38, 43, 58, 59, 62], or be hard to configure [43, 58, 62, 65] and integrate into existing workflows [8, 23, 51]. These usability barriers often stem from mismatches with developers' workflows and inaccurate mental models of how the tools operate, which can lead to distrust and abandonment [46, 61].

Rangnau et al. studied dynamic testing beyond fuzzing, focusing on its integration into developers' workflows [52]. Mattei et al. performed a heuristic evaluation of 288 security tools (including fuzzers), assessing their expected usability [40]. They found that most security tools provide limited interaction and usability support, with dynamic tools in particular exhibiting readability challenges.

Some researchers have worked to develop more human-centric interfaces for security tools. Katcher et al. used paper prototypes to identify user needs and provide interface design recommendations for automating protocol reverse engineering [24]. Yakdan et al. developed a decompiler designed for code readability and showed that it improved program understanding [72]. These successes motivate our effort to characterize usability challenges throughout the lifecycle of real-world fuzzing campaigns and identify potential human-centric design improvements.

3 Method

To explore our research questions, we conducted semi-structured interviews with security professionals, software developers, and security researchers who have applied experience using fuzzers in academic or industry settings. Our goal was to understand how these professionals interact with fuzzers, challenges they encounter, and strategies they employ to improve their workflows.

3.1 Recruitment

We recruited 18 participants based on their experience with fuzzers. To screen for relevant experience, we sent potential participants an initial survey that collected demographic and contextual information, including professional background, years of fuzzing experience, and the specific tools they used. The survey responses helped to identify participants with substantial fuzzing experience and provided context for interpreting their interviews.

As prior work has shown a learning curve for fuzzing, we included participants with diverse experience levels to capture different perspectives along this spectrum. To focus on fuzzer usage, we excluded potential participants whose primary expertise was in fuzzer *development*. Instead, we selected for experience in areas such as automated harness generation, fuzzer benchmarking, and general software security, all of which involve practical tool usage.

Recruited participants from all backgrounds predominantly used mainstream grey-box fuzzers (AFL, libFuzzer, and LibAFL), including potential participants who built custom wrappers around these tools [39]. Consequently, while we did not set out to focus on a

specific fuzzer category, our study's findings organically centered on the real-world use of these widely adopted tools, allowing us to capture a comprehensive picture of their common challenges and strengths.

We recruited academic and industry participants,[1] using the following strategies to ensure a diverse and qualified sample:

- **Public contributions**: We identified GitHub users who had submitted fuzzing-generated test cases for vulnerabilities in C, C++, or Rust programs. These submissions demonstrated practical expertise and real-world fuzzing success. When GitHub profiles included a personal website with contact information, we reached out directly to invite participation.
- **Academic publications**: We contacted authors of recent research papers that involve applying fuzzers. Our goal was to recruit researchers with hands-on experience using fuzzing in varied contexts, while excluding those whose primary experience was limited to developing fuzzers.
- **Online fuzzing communities**: We posted recruitment messages in multiple fuzzing-related Discord servers. These communities include practitioners, researchers, and hobbyists, helping us reach a broader pool of participants.
- **Professional networks and snowball sampling**: We recruited additional participants through personal networks and referrals, including asking participants to recommend colleagues or peers with relevant experience.

3.2 Interview protocol

We conducted 18 semi-structured interviews between October 2024 and April 2025, each lasting approximately one hour. Each interview followed a semi-structured format, allowing us to cover key topics while also adapting to participants' unique experiences. All interviews were conducted over Zoom, except for one which was conducted via Discord due to the participant's preference.

This study was approved by the University of Maryland's Institutional Review Board. Participants were informed about the purpose and scope of the study prior to participation through a consent form at the beginning of the screening survey, where they confirmed their agreement to proceed. We verbally confirmed consent again at the beginning of each interview. Participants were compensated $75 USD for their time. Only audio from the interviews was recorded, with participants' permission. Recordings were stored on secure, access-controlled machines. We transcribed interviews using OpenAI Whisper [47], an automated service. The model was run entirely on local machines. Identifiable information was removed or anonymized during transcription and analysis. Although some participants discussed professional or project-related tooling, we omit proprietary and sensitive technical details from our findings.

The interviews were structured around three key topics, reflecting major stages of the fuzzing workflow discussed in Section 2.2:

- **Fuzzer setup and usage:** How participants select and prepare fuzzing targets, including instrumentation, harness creation, seed selection, and configuration.
- **Monitoring, interpreting, and managing output:** How participants analyze and act on fuzzer findings, addressing challenges such as redundant reports, prioritization strategies, and extracting actionable insights.
- **Opportunities for improvement:** Reflections on potential enhancements to fuzzers, including desired features, usability improvements, and the integration of fuzzing with other security testing methods.

While the interview script provided structure, we adopted an iterative approach. As themes emerged, we refined or changed our follow-up questions to probe relevant challenges in more depth. We also updated our screening criteria: for example, once we reached saturation with participants who primarily conducted fuzzer evaluations, we began filtering out individuals with similar backgrounds. We continued conducting interviews until we reached thematic saturation. The research team determined saturation through regular discussions of emerging themes after each interview. We observed that novel insights began to diminish after the fifteenth interview, and the final three interviews confirmed that our data had stabilized.

3.3 Data analysis

We conducted a collaborative thematic analysis to identify recurring themes and insights [9], through the following stages:

(1) **Collaborative initial coding:** Two researchers jointly reviewed a subset of transcripts, coding them together to develop a preliminary codebook informed by early patterns in the data.
(2) **Double-coding and refinement:** The two researchers independently double-coded several additional transcripts, meeting regularly to discuss differences, resolve discrepancies, and iteratively refine the codebook.
(3) **Primary coding with review:** The remaining transcripts were coded by one researcher, with the other providing review and feedback. Throughout this stage, we continued to adjust and refine the codebook, holding regular meetings to ensure consistency and resolve uncertainties.

This collaborative coding process ensured the analysis was reliable and comprehensive, combining multiple perspectives to enhance the depth of the findings. We did not calculate inter-rater reliability, consistent with interpretivist approaches to qualitative research that emphasize collaborative sensemaking over statistical agreement [41]. The final codes informed the analysis of key themes, which are presented in Section 4.

3.4 Limitations

While our study includes participants with a range of fuzzing experience, we focus on widely used tools such as AFL/AFL++, libFuzzer, and their variants. These tools are the most common entry points into fuzzing and come up frequently in both academic research and practical security engineering. While we emphasize themes that transcend specific tooling, we may capture usability patterns and challenges most representative of common practice, at the potential cost of underrepresenting experiences with niche fuzzers or highly specialized domains.

Another potential limitation is self-selection bias. Participants who responded to our outreach likely have stronger interest in fuzzing and improving fuzzing workflows, and may thus have more

[1] Some participants based in academia shared insights drawn from previous or ongoing industry collaborations.

interest or experience in overcoming tooling challenges. We may miss some challenges experienced, for example, by practitioners who tried but abandoned fuzzing due to poor initial experiences. To partially account for this, we included questions focused on learning barriers, early-stage struggles, and moments of confusion.

Interview-based studies can introduce biases: participants may misremember past experiences, and interviewer perspectives may shape interpretations. To mitigate this, we asked for concrete examples, used a semi-structured protocol, and relied on collaborative coding and iterative analysis [54].

Despite these limitations, our study offers practical insights into the challenges and opportunities of using fuzzers, grounded in diverse real-world experiences.

4 Results

In this section, we present our findings, organized into five categories. We begin with a description of our participants to contextualize their experiences. Next, we describe participants' mental models of fuzzers. We then present challenges, practices, and suggestions for improvement across three stages of the fuzzing workflow: (1) preparing targets and configuring fuzzers, (2) running and monitoring fuzzing campaigns, and (3) integrating and extending fuzzing within broader development and testing workflows. While we occasionally report participant counts to provide context for specific themes, these numbers are not intended to indicate prevalence in the broader population.

4.1 Participants

Table 1 summarizes participants' experience, organizational context, and fuzzer usage. We categorize fuzzer use cases based on participants' roles and goals. Participants often have multiple use cases, which we capture across the following major categories:

- Software development: Using fuzzers as part of regular development workflows to catch bugs early.
- Cybersecurity: Applying fuzzing to discover vulnerabilities or assess the security posture of software systems.
- DevOps: Using fuzzers as part of CI/CD pipelines to ensure stability issues are caught continuously.
- QA/testing: Using fuzzers within broader software testing strategies, focusing on software from third parties.
- Research: Employing fuzzing as part of academic or industry research projects, such as tool evaluation, network protocol, or auto harness generation.

Geographically, 12 of the 18 participants were based in the United States, while the remaining six were located in other countries across Europe and Asia.

4.2 Participants' understanding of fuzzers

While all study participants are experienced users of fuzzers, and many have deep technical knowledge, including research backgrounds, their understanding of how fuzzers work varies widely. Some participants indicated highly developed understandings of fuzzing internals, while others expressed uncertainty about how fuzzers work "under the hood." This variation often reflects the informal, self-directed nature of how they learned to use fuzzers.

Fuzzing is uniquely valuable. Participants consistently expressed strong enthusiasm for fuzzing—not just as a practical testing technique, but also as a foundational concept in their approach to understanding software behavior. Even when fuzzers do not fully align with their workflows or goals, participants see fuzzing as indispensable for surfacing bugs other methods might miss. Rather than viewing it as just another automation tool, many regard fuzzing as a fundamentally different way of reasoning about software reliability that emphasizes unpredictability, emergence, and exploration.

P3, for instance, described fuzzing as something developers "are missing out on," stressing that while unit tests cover expected cases, "to find things that cannot be predicted, fuzzing is definitely the *de facto* thing to do." P18 echoed this perspective, saying, "I realized [fuzzing] is a really good testing technique. . . . Why are developers not utilizing that enough? It is really powerful, extremely powerful." Participants who had embraced fuzzing often reported a shift in their testing mindset: "Once you get past the learning curve, you're entering a whole new realm of testing. You'd never fully trust test cases anymore—you'd only trust whether something is bug-free after running a fuzzer on it" (P3).

In addition to applying fuzzing for bug detection, participants have adopted it as a core mental model for interrogating uncertainty in complex software systems, cementing fuzzing as indispensable.

Fuzzers are worth adapting, even beyond their ideal scope. While many consider fuzzing indispensable, participants recognize fuzzing is not currently universally applicable, especially with limited tool sets. As P12 put it, "Currently, fuzzing is only a great fit for a few problem domains . . . typically, file formats, network-based formats, parsing, decoding, deserializing." These input-driven, structurally predictable systems are widely seen as well suited to fuzzing compared to software with complex state or semantic logic.

Rather than abandoning the technique when not directly applicable, participants frequently find ways to adapt fuzzing to new contexts by applying it as a general-purpose probe for problems such as understanding how a legacy system behaves, testing semantic consistency between components, and monitoring coverage in machine learning frameworks. More broadly, participants described applying fuzzing to surface edge cases, reveal implicit assumptions, or provide behavioral baselines that inform further manual testing.

Despite the fact that adapting fuzzing to these contexts often requires additional effort (setup, scripting, or output reinterpretation), participants characterized fuzzers as uniquely capable and worth the effort, even when imperfectly aligned with their immediate goals.

As black boxes, fuzzers create limited trust and interpretability. While participants acknowledged the strong capabilities of fuzzing, they also often described fuzzers' internal behavior as difficult to reason about. While all participants indicated an understanding of how fuzzers *should* work, they find that real-world use often offers little feedback about what happens internally, resulting in unpredictability and opacity. As P3 put it, "Even if you do a deep dive into the fuzzer, it's always going to feel like a black box."

Participants often rely on surface-level metrics such as line or path coverage, crash counts, or timeouts to gauge progress. While

ID	Role	Exp.	Tools	Org. type	Fuzzer use cases				
					Software dev.	Cybersecurity	DevOps	QA/testing	Research
P1	Software engineer	1–2 yrs	●, △, ★	Large	✓			✓	✓
P2	Research assistant	1–2 yrs	●, △	Research		✓			✓
P3	Research assistant	1–2 yrs	●	Research					✓
P4	Software engineer	>5 yrs	●, △, ★, ◇	Large	✓	✓			✓
P5	Research assistant	3–5 yrs	●, △	Research		✓			✓
P6	Research assistant	1–2 yrs	●, △, ∇, ◇, ★	Research	✓	✓	✓		✓
P7	Security engineer	>5 yrs	●, △, ★	Large	✓				✓
P8	Security researcher	<1 yr	●, ∇	Research	✓		✓	✓	
P9	Research assistant	1–2 yrs	●, △, ∇	Research	✓			✓	✓
P10	Research assistant	3–5 yrs	●, ★	Research		✓			✓
P11	Software engineer	>5 yrs	●, △, ∇	Large	✓	✓			
P12	Security engineer	3–5 yrs	●, △, ★	Small/Med	✓	✓		✓	
P13	Research assistant	3–5 yrs	●, ◇	Research	✓				✓
P14	Security researcher	>5 yrs	●, △, ★	Large		✓			✓
P15	Security engineer	>5 yrs	●, △	Small/Med	✓			✓	
P16	Security engineer	3–5 yrs	△	Small/Med	✓				
P17	Research assistant	3–5 yrs	●, △, ★	Research	✓				✓
P18	Security engineer	3–5 yrs	●, △	Large	✓	✓	✓		

Table 1: Demographic details of interview participants. Tools: ● AFL, △ LibFuzzer, ∇ Honggfuzz, ◇ PeachFuzzer, ★ OSSFuzz. Org. type: Large = 500+ employees; Small/Med = <500; Research = university/industry research.

they recognized these metrics were imperfect, they are often the only feedback available. As P4 explained, "We don't really have a clear way to determine, for example, what the absolute performance of a fuzzer is or how to fully evaluate it. That lack of a ground truth makes it challenging to assess fuzzers." Others, like P12, described relying on intuition or anecdotal experience rather than measurable signals: "Just my own personal experience, AFL has been better at finding bugs than LibFuzzer. . . . I don't have data to back that up . . . just my own personal experience." This uncertainty further contributes to the perception of fuzzers as black boxes, even among users with deep technical experience. We discuss participants' strategies for dealing with this challenge in Section 4.4.

4.3 Configuring fuzzers and preparing targets

Before fuzzing can begin, users must prepare both the fuzzing target and the fuzzer. Here, we describe the challenges participants face in doing so, and how these challenges impact their ability to use fuzzers effectively. We then highlight participants' current strategies for choosing appropriate tools and iterating toward workable setups. Finally, we present participants' ideas for improving this workflow stage, including more automation, modularity, and support for broader problem domains. We note that not all participant suggestions (in this section and later) are necessarily realistic given the fundamental nature of fuzzing, but they do highlight desired user experiences. Going forward, we use the labels **Challenge**, **Practice**, and **Suggestion** to signal the focus of each finding.

Challenge: Due in part to limited learning resources, participants feel they have inconsistent and narrow understanding. A root cause of many challenges is participants' strategy of learning how to use fuzzers on the fly, which often limits the generalizability of their knowledge across tools and contexts. Most participants described learning fuzzing through hands-on practice (11/18) and/or written resources such as blogs, academic papers, and documentation (13/18). While this approach often worked for quickly setting up a specific fuzzer, it also led to perceived gaps or inconsistencies in conceptual understanding, especially when reasoning about tool behavior, effectiveness, or advanced feature configuration. As P5 put it, learning by "reading blog posts and experimenting [made it] hard to tell if I'm doing it right."

When documentation or tutorials fail to provide sufficient guidance, some participants (6/18) reported turning to fuzzers' source code to understand how specific components work or how to extend the tool. For example, P3 said, "If my initial searches aren't helpful, I would jump into the code of the fuzzer itself." This can deepen users' knowledge about the fuzzer, but this knowledge is tied to a specific tool's implementation rather than general principles.

Participants noted that the absence of centralized, structured, or pedagogically designed learning resources—on top of fuzzing's already steep learning curve—makes it difficult to build a coherent understanding that transfers across tools and workflows. Even technically advanced participants reported relying on ad hoc sources and personal heuristics. This complicates onboarding for new users and introduces friction when sharing workflows, debugging issues collaboratively, or evaluating fuzzing effectiveness in a reproducible way.

Many participants commented on this limited generalizability of knowledge when switching fuzzers. P14 noted that "if you're working on a new fuzzer and you want to reproduce the results of another fuzzer, even a small change in configuration can lead to completely different results." Underscoring the cost, P6 said, "The time wasted trying to make things work again in another new fuzzer is huge," noting that different fuzzer architectures require different libraries and knowledge. P18 further highlighted that fuzzing

techniques vary dramatically, contrasting web application directory fuzzing with coverage-guided fuzzing, where "you're actually instrumenting the code and trying to increase the code blocks coverage." P14 described how even experienced practitioners struggle with tool-specific practices: "They [coworkers] often ask how to instrument a program properly because a lot of techniques require modifying the compiler. … Each fuzzer has its own instrumentation approach." Together, these accounts show that some knowledge acquired in specific one fuzzer or context rarely transfers cleanly to another, forcing practitioners to repeatedly reverse engineer best practices rather than build on shared or standardized guidance.

Challenge: Setup is confusing and poorly aligned with users' needs. To help with setup, most fuzzers come with documentation, Docker containers, or tutorials. However, participants said these are often not detailed enough, leading to struggles and a cycle of trial and error with configuration flags, instrumentation steps, and tool-specific adjustments. As P3 explained, "There's a lot I wasn't familiar with … especially the flags. There are so many, and I wasn't always sure whether they were doing what I wanted them to do." Furthermore, as P17 reflected, "A lot of the documentation … requires a ton of prior knowledge that a lot of people don't have," creating a significant barrier to entry.

Participants also pointed out that fuzzers are often only available or optimized for certain use cases (e.g., small C/C++ command-line utilities) that do not reflect modern software's structure and interfaces. P18 explained, "Fuzzing needs to be applied to more different platforms. We don't deal with x86 only; we have different platforms and IoT devices that you need to write software for."

Participants noted that adjusting flags or compiler options to improve fuzzer performance must be done carefully to avoid unintended consequences. As P4 warned, "Setting up the fuzzer and configuring it appropriately is something we need to be very careful with. … Certain [compiler] optimizers might reduce the sensitivity of the fuzzer." As a result, some participants deployed fuzzers using default settings, even though this did not align with their needs.

Challenge: Complex fuzzing targets impact the ability to apply fuzzing effectively. Beyond setup, participants frequently encounter challenges preparing real-world programs for fuzzing, especially for large, layered, or legacy codebases. Many of these systems are poorly documented, difficult to build, or rely on outdated dependencies. As P14 put it, "Some of these projects were written 30 years ago—you never know what to expect."

These problems were magnified by the technical requirements of coverage-guided fuzzers. Many require recompiling the fuzzing target with custom instrumentation compilers (e.g., AFL++'s modified Clang or LLVM pass) to provide metrics and guide input generation. But, as P14 noted, these extra steps "may fail if you enabled certain optimization flags, or even fail by itself," since "open-source and legacy programs can be surprisingly fragile." Further, as P9 mentioned, "Different programs are built differently," and "understanding where to introduce the AFL++ compiler in that build process is the tricky part."

Challenge: Identifying targets in unfamiliar codebases. Even after a successful build, participants reported lacking principled strategies for selecting where to fuzz, especially when they were fuzzing unfamiliar code. Instead, they rely on intuition, heuristics like choosing functions that "looked interesting" (P16), or using commercial tooling to surface likely targets but then applying personal judgment to finalize the decision. P15 said they "just identify functions that look like they could be good targets."

These uncertainties surrounding the program under test often compound participants' existing uncertainty about fuzzers. Participants reported being unsure how to apply their high-level fuzzing knowledge to large, poorly understood, or legacy systems. This mismatch between theoretical knowledge and practice introduced hesitation and limited participants' confidence in their strategies.

Practice: An iterative approach toward a working setup. To cope with setup challenges, participants prioritize getting their chosen fuzzer up and running with minimal friction. Rather than aiming for a perfect setup from the start, they treat configuration as an iterative, trial-and-error process: run the tool, observe what breaks, adjust the harness or flags, and try again. During this process, participants value speed and feedback over precision: even though early runs are expected to be shallow or error-prone, participants see value in getting something running quickly in order to iterate and learn from results. As P16 explained, "For a first pass, my priorities are to get something running as fast as possible."

To reduce complexity and improve reliability during early stages of fuzzing, participants often seek to simplify their environment. Some participants (4/18) emphasized the value of isolating the fuzzer to prevent interference from the host system or conflicting software. As P10 explained, "I usually run fuzzing campaigns in a Docker container … so it has its own space, its own memory, and so on." In this vein, participants may also start with minimal harnesses, often "feed[ing] the fuzzer input to whatever function in the API looks most top-level" (P16). These lightweight setups allow participants to validate that the tool is functioning before adding complexity.

Practice: Selecting fuzzing targets based on experience, intuition, and documentation. A critical part of setup is identifying where to fuzz within a fuzzing target. As noted earlier, participants often lack principled strategies for target selection and default to intuition. Even when external tools are available to suggest candidate entry points, participants still shoulder much of the cognitive load when determining which to fuzz, usually relying on their understanding and intuition. As P15 reflected, "Even with help… the way we select functions is not scientific. It's intuition-based."

To gain a deeper understanding of an unfamiliar fuzzing target, participants use documentation and example code to help identify stable or representative entry points. As P17 expressed, "If I'm trying to harness a function that's in the example documentation, I can look at that."

When fuzzing workflows require or would benefit from initial seeds, most participants reported extracting them directly from the fuzzing target. A common strategy is to repurpose unit test inputs or files from the project's test suite. P18 considers this the "best way" to obtain seeds, since developers "know their code the best."

Suggestion: Fuzzing setup should be automated and guided. As a result of the setup challenges described above, participants

expressed a strong desire for tooling to guide or automate the setup process, including help to identify viable targets, generate harness scaffolds, and configure initial parameters efficiently.

The most frequently requested feature was automatic harness generation. As described earlier, writing a harness requires understanding the fuzzing target—often involving time-consuming, manual inspection of unfamiliar code. To reduce this burden, participants want tools that could automatically generate runnable harnesses based on program structure or existing usage patterns. As P2 noted, automated harness generation would mean that "I don't need to write my own harness or specify the command line manually. It could generate something the fuzzer can run."

Others emphasized the potential of integrating fuzzing directly into development environments. For example, both P12 and P15 envisioned a possible Visual Studio Code plugin that could automate common setup steps: "A plugin that looks for targets, suggests a harness with a large language model, writes a test file, and then you just click go—you're fuzzing." For participants like P12 and P15, seamless IDE integration could significantly reduce the barrier to starting and iterating on fuzzing tasks.

To further streamline setup, participants want assistance identifying good entry points and configuring fuzzer parameters. Currently, participants said they often rely on intuition or partial static analysis, but they believe these tasks could—and should—be supported more directly by tooling. P9 described their ideal feature set as "automatic harness generation, automatic instrumentation, and automatically figuring out which configuration flags are best." Similarly, P15 suggested static analysis could be used to generate, filter, and rank a list of fuzzable functions, reducing the need to "manually go through every function."

Even after setup, participants find it difficult to assess whether their configuration choices, such as seed selection or entry point targeting, are likely to yield useful results. Several expressed interest in tools that could provide early indicators of setup quality before a campaign begins. As P10 proposed, "Something like a metric from previous fuzzing campaigns … that measures the quality of the initial seed" would help users iterate more strategically.

Participants independently suggested AI as one possibility to simplify or guide the setup process. While few had used large language models directly for fuzzer configuration, several suggested it might be useful. P14 imagined a future where GPT-like tools could "read the files, handle errors, fix the command iteratively. … That would make life much easier." This aligns with emerging research exploring AI-assisted harness generation and automated fuzzing workflows, which we briefly discussed in 2.2.

Together, these suggestions reflect a desire to move beyond low-level configuration and toward more intelligent, goal-driven fuzzing setup, with tools that can reason about context and recommend strategies, helping users get started with less trial and error.

4.4 Monitoring fuzzing campaigns

Once a fuzzer is running, users make ongoing decisions about how to monitor progress, how to interpret output, and how long to continue. In this section, we describe the challenges participants encounter during active fuzzing, including difficulties interpreting fuzzer output and uncertainty about when to stop. We highlight the practical strategies participants use to improve efficiency and gain insight, such as active observation and manual output triage. Finally, we present their suggestions for improving this stage of the workflow, including calls for more actionable output, clearer signals about progress, and better support for analyzing and reproducing crash results.

Challenge: Crash output is redundant and unhelpful. While they do not expect crash reports to serve as a complete diagnosis, participants emphasized that outputs often lack sufficient context to understand the actual issue or severity. When asked whether AFL's output was helpful for pinpointing bugs, P9 described it as "more of a starter. I take that input and use other tools to figure out where the bug is. … I would re-execute it to see if that problem persists; then I inspect it with memory analysis tools." This response, echoed by others, highlights that fuzzing results require considerable post-processing and external analysis to become actionable.

Participants reported large volumes of redundant crashes that require manual inspection. P3, who tests complex libraries, described the scale of this problem: "Out of those thousand crashes, 900 of them can be redundant—it becomes really hard to manually go through each and every one of them. That's fundamentally a challenge."

While some tools offer deduplication or grouping to minimize redundancy, participants often expressed distrust of these features, fearing that unique bugs might be incorrectly grouped or silently discarded. P7 said that AFL's build-in deduplication sometimes "clusters different crashes into the same group. And sometimes it clusters one crash incorrectly into a different group. We have to fine-tune it really carefully in terms of which heuristics we're going to apply." P11 had experienced the opposite problem with AFL's deduplication: "We still found cases where different crashes actually stemmed from the same bug but were recorded as distinct."

In addition to crash reports, fuzzers produce timeout reports; participants also find these difficult to interpret. While many consider timeouts a potentially valuable signal—often indicating hangs or performance bottlenecks—fuzzers rarely provide enough context to evaluate their cause or severity. As P6 explained, "Does a timeout mean there was a bug? Or could there be multiple bugs? How do you interpret the timeout? It becomes very nuanced." This ambiguity leaves participants uncertain whether to treat timeouts as bugs worth triaging or simply as noise.

False positives further compound uncertainties in output analysis. Participants use the term differently, reflecting fuzzing's varied contexts and goals. For some, false positives referred to redundancy: P1 equates them with duplicate crashes, emphasizing the burden of triage and tooling efficiency, while P3 considers false positives unique bugs that are mistakenly discarded. Others used the term to describe inputs that do not correspond to real vulnerabilities, as P13 explained: "We will have many false positives ... but it's actually not a vulnerability." P17 highlighted another dimension, emphasizing their focus on harness setup: "False positive crashes could be occurring from my harness itself." In contrast, when asked, P7 downplayed the importance of false positives: "It is good enough we have a bug to report. I don't have time to worry about anything else." This diversity of considerations illustrates why fuzzer output

is particularly difficult to act on: practitioners must constantly evaluate results against their own definitions of correctness, severity, and root cause.

Challenge: Runtime outputs lack key details and context. Beyond crash output, participants also expressed frustration with output produced while fuzzers are actively running. Many said it is difficult to tell whether their campaign is making progress. While some fuzzers expose performance metrics like coverage growth and seed queue size, these metrics are often cryptic, under-documented, or omitted entirely. As P12 said, "Runtime outputs are famously inscrutable. … When you look at AFL's output or libFuzzer, it's all just abbreviations of things. Nobody knows what the heck they are until you read the source code."

Participants who had attempted to inspect logs during fuzzing found that crucial contextual information is often missing. For example, P11 noted that "it doesn't always log the parent seed, which mutation operand was used to generate it, or even the exact time of discovery." Without this information, participants find it difficult to trace how a particular input was derived, assess why it triggered a bug, or reproduce conditions under which a crash occurred.

This lack of transparency is exacerbated by the inherent randomness of fuzzing. As P14 explained, "Fuzzing results are inherently random. A fuzzer might find a bug in one run but miss it in another." Without detailed metadata about input provenance and mutation history, participants struggle with performance gap analysis and root cause identification, making it harder to diagnose inconsistent behavior or optimize future runs.

Taken together, these issues point to a core usability gap: fuzzers can generate a lot of output, but it is often not readily informative. Participants have to exert significant effort to make sense of crashes, triage results, and assess campaign effectiveness.

Challenge: Stopping criteria are not well defined. Fuzzing is inherently an open-ended process, and participants understand that new bugs can always be found with more time or better mutations. However, real-world constraints like compute budgets and project deadlines mean they eventually have to decide when to stop. Making that decision, however, is far from straightforward.

Some participants (5/18) explicitly mentioned that they rely purely on intuition or informal heuristics. As P15 put it, "There's no rigorous 'when fuzzing is done' criteria. … I'll typically set up a fuzzing harness, run it overnight, and if in the morning there are no finds, I'll say, 'Okay, there's nothing here to be found.'"

Others have tried to use metrics like branch or path coverage to guide the decision. But even this approach is fraught with doubt, especially when participants are unsure whether stagnation reflects a truly exhausted search space or a bug in their setup. As P18 described, "If the coverage is not increasing anymore … maybe there's a bug in our harness."

Ultimately, many participants default to setting arbitrary time windows—ranging from a few days to a few weeks—based on practical constraints (e.g., perceived progress or resources) rather than technical indicators. P18 explained, "A couple of days at least. Maybe like a couple of weeks even. We don't have like a specific defined time window, but we can decide to stop if this fuzzing process is consuming a lot of resources." Some participants (10/18), particularly those in research contexts, have adopted 24 hours as a standard campaign length, not because it is ideal, but because it aligns with common benchmarking practices.

While participants do not expect fuzzers to offer a definitive "you're done" signal, they consistently expressed a desire for more meaningful feedback about progress. The broader challenge is not necessarily about finding a stopping rule, but rather about dealing with the fundamental uncertainties of fuzzing in more informed ways.

Practice: Monitor early signals and prioritize efficiency. To compensate for limited runtime feedback and unclear stopping criteria, participants have adopted practical strategies to improve the efficiency of their fuzzing campaigns. Rather than relying solely on built-in metrics or default configurations, they emphasize active monitoring and workflow-level optimizations to detect problems early and maximize returns.

Several participants (5/18) highlighted the importance of directly observing fuzzer behavior during the early stages of a campaign, allowing them to catch setup problems early. As P1 put it, "It's important to observe the process in real time rather than just starting the fuzzer and walking away." Similarly, P9 described using early coverage metrics as a sanity check: "What I'm usually looking for is if there's absolutely no new branches found in the first couple of minutes." Participants emphasized that direct monitoring is especially important with limited tool support for runtime diagnostics and termination guidance. In practice, participants rely on a small set of interpretable signals, such as coverage and execution speed, to assess fuzzing progress.

Others turn to parallelism to maximize fuzzer efficiency and returns. Some run multiple fuzzers or instances of the same fuzzer with synchronized corpora to increase path discovery. P14 combines these approaches, explaining, "I would run three or four instances of the same fuzzer that synchronize their corpora periodically. Additionally, I would parallelize different fuzzers."

Practice: Start with deduplication, then manual triage. Participants follow a common pattern when analyzing fuzzer output: first deduplicate crashes, then manually investigate those that remain. Many rely on custom scripts that go beyond the provided fuzzer functionality in order to cluster or filter crashes based on stack traces, memory addresses, or observed behavior. This initial pruning step is seen as essential to reduce noise and focus attention on distinct issues.

After deduplication, participants shift to more in-depth analysis, such as rerunning crashing inputs, tracing execution using tools like GDB or Valgrind, or inspecting code to identify root causes. While time-consuming, this manual process is seen as necessary to understand the significance of each crash and determine whether it indicates a real vulnerability.

Participants also shared creative adaptations to streamline this process. For example, P3 described an ad-hoc sampling strategy: "Rerun [crash reports] … and randomly sample the outputs. Once I see a pattern in the outputs, I create a script to prune out those specific patterns." This approach deviates from tool-provided heuristics,

reflecting the extent to which participants have to rely on personal judgment and scripting to make outputs manageable. While these techniques are seen as making analysis more tractable, they are also labor-intensive and require judgment and scripting skills. P17 employs a different strategy to streamline manual analysis: "In some cases, you can look it up online and try to see if other people experienced a similar crash."

Suggestion: Provide detailed, actionable runtime output. As noted above, participants consistently expressed frustration with insufficiently detailed runtime output, which makes it difficult to debug fuzzer setups and adjust strategies during long-running campaigns.

Several participants (6/18) emphasized the need for richer, real-time feedback during fuzzer execution. Rather than waiting until the end of a run, they want visibility into the internal behavior of the fuzzer while it is running—such as which paths are being hit, how frequently, and where progress is stalling. As P8 expressed, "We want to get more insights into why a fuzzer is having difficulties finding a particular bug. … If we could know the frequency with which the fuzzer hits each path, that would be helpful."

Besides richer runtime feedback, some participants (6/18) envisioned visual interfaces that could make fuzzing dynamics more interpretable, especially for understanding execution paths and bottlenecks. P10 imagined a web-based dashboard that would overlay program structure with fuzzing behavior: "With the source code on one side and a function-level view of how fuzzing is working … you could see what inputs passed through which functions and visualize the execution tree."

A few participants (4/18) discussed possible human-in-the-loop features as a promising direction. Rather than treating fuzzers as black boxes, participants want a tool that could "recommend how I can fuzz better, like offering tips or strategies to improve the fuzzing process" (P3). This collaborative tool could "tell me when it's no longer being efficient" or "maybe even recommend stopping or fuzzing again later when the code has changed significantly" (P11). These techniques would allow users to guide or interact with the fuzzer mid-execution, by flagging bottlenecks or injecting hints, instead of simply relying on automated exploration.

Suggestion: Make crash output easier to triage and reproduce. In addition to runtime feedback, participants agreed that fuzzers need better support for crash triage and reproduction. As described earlier, built-in dededuplication approaches leave room for improvement, often requiring participants to develop custom solutions.

Beyond deduplication, participants want crash output to be more actionable. Some participants (4/18) suggested smoother reproduction mechanisms, such as the ability to "inject crashing inputs directly into the target" (P1) or analysis hooks similar to those provided by tools like GDB, highlighting that this would help reduce the overhead of re-executing crashes. Others (4/18) suggested that fuzzers should provide flexible and informative categorization of crash types. Specifically, they want tools to "automatically classify the different results … based on the kind of bugs detected" (P2) to help users prioritize and understand outcomes more effectively.

These suggestions reflect a broader desire for fuzzers that find failures *and* support users in diagnosing and addressing them with less manual effort.

4.5 Integration of fuzzing workflows

Beyond setup and runtime, participants often need to adapt fuzzers to fit broader development workflows or support more advanced testing goals. In this section, we describe the limitations participants have encountered when applying advanced features, customizing tool behavior, or scaling fuzzing across large systems. We outline the workarounds participants have used to integrate fuzzers into their pipelines—often writing scripts or wrappers to make tools fit their specific context. Finally, we present suggestions for improving this stage, including better user interfaces, more flexible architectures, and features that help users coordinate fuzzing with other development and testing activities.

Challenge: Fuzzers are overcomplicated and hard to adapt. While modern fuzzers offer powerful capabilities, many participants find them over-engineered, difficult to configure, and poorly aligned with practical workflows. P4 noted that "the community has been merging a lot of tools and mods into single platforms," which adds complexity and makes tools harder to work with. Several participants pointed out that setups intended to simplify fuzzing, such as Docker-based workflows, sometimes introduce unnecessary complexity. As P7 described, "They had one Docker image to build the fuzzer, a second to compile the target, and a third to actually run it." What was meant to streamline experimentation instead created a set of complicated pipelines that are difficult to modify.

Several participants noted that without usable, transparent interfaces, new fuzzers that are theoretically better for a given task may not get adopted. P3 mentioned attempting and failing to use a specialized tool that was designed for their intended use case: "If I had a better interface to understand what's going on … I could have leveraged (the tool) better. But it was so noisy. … I just fell back to AFL."

Recent innovations like hybrid fuzzing, which combines fuzzing with static or symbolic analysis, aim to promote usability by increasing code coverage and automating deeper bug discovery. Ironically, participants (6/18) described them as especially difficult to configure and fit into their workflow. As P8 explained, "They normally have a symbolic executor running in parallel, and I need to figure out the steps to set up both parts." Some tools run multiple analysis modes simultaneously, which makes it difficult to manage or control fuzzing sessions, in turn making it difficult to "figure out how to stop them once they found the first crash" (P8).

Participants working in large-scale environments raised additional concerns about performance tuning. Despite having high compute capacity, they had found that tools do not scale well and require manual duplication of tasks to avoid performance degradation. P11 explained, "On our system with 128 cores, running 64 threads often performs worse than 32. … We introduce duplicate fuzzing tasks to prevent degradation." These workarounds are not just technical annoyances; they also introduce inefficiencies that drain resources and limit the feasibility of deploying fuzzers consistently at scale.

Together, these accounts reveal a growing disconnect between the increasing complexity of modern fuzzers and the practical needs of those who use them.

Challenge: Research prioritizes performance. Several participants (5/18), particularly from industry, expressed frustration with the broader culture of fuzzing research, which they see as overly focused on achieving marginal gains in technical metrics rather than improving usability or addressing practical barriers to adoption. Specifically, participants contrasted research on advanced mutation strategies, hybrid analysis, and deep state exploration with a lack of attention to usability, such as clearer configuration interfaces, modularizing components, and improved crash triage.

P12 observed, "I think fuzzer capability is good enough … but I don't see a lot of investigation into how to make these fuzzers easier to use." This view was echoed by P17, who considers many advancements in fuzzer performance irrelevant to their work: "There can always be changes made to the efficiency … but those are things that I'm not particularly concerned [about]."

Participants across both academia and industry emphasized that real-world adoption depends on usability. However, P14, who works in industry, contrasted the two: "Industry cares more about usability than academia does. … When we release a new fuzzer, we want people to use it. In industry, if a tool isn't easy to use, it won't get adopted."

Challenge: Lack of standardization and reproducibility. Participants also criticized a perceived lack of standardization and rigor in fuzzer evaluation.[2] P14 noted that despite efforts like FuzzBench [1], "There's no standardized way to evaluate. …In this environment, a bad baseline is a good baseline. That's a major problem in fuzzing research." Without agreed-upon benchmarks and consistent setups, participants from both academia and industry expressed skepticism about how well academic fuzzing results translate into practice.

Beyond research settings, lack of standardization also makes collaboration and reproducibility more difficult. Differences in architecture, environments, and dependency versions create headaches when sharing setups or reproducing results. As P6 noted, "Certain combinations just don't work. …On a different architecture, you might need a different version of a library that wasn't necessarily packaged with the program you're trying to use. You can do all your due diligence … but as soon as a couple of things change … the next person who tries to reproduce your work often has to invent a way to make it work." They estimated that the cumulative cost of this brittleness—especially in complex, low-level systems code—likely amounts to "millions of dollars" in wasted developer time.

These concerns are not limited to academia. Participants noted that when state-of-the-art research lacks standardization and robust evaluation practices, it becomes difficult for industry to benchmark tools, assess tradeoffs, or justify adoption, limiting the deployment of fuzzing at scale.

Practice: Selecting and adapting the right tool(s) for the task. To better integrate fuzzing into their workflows, participants select and adapt tools based on the specific demands of their testing goals, software constraints, and resource availability. While some fuzzers attempt to offer general-purpose functionality across domains, participants agreed that no single fuzzer is universally effective. Instead, they make pragmatic choices about which tools to use for specific contexts, such as targeting compiled binaries, specific APIs, or protocol-level behavior. These tool selection decisions are also shaped by practical concerns such as deadlines, familiarity with tooling, and the complexity of the fuzzing target.

Many participants (8/18) paired fuzzing with other analysis techniques to improve effectiveness. P12 described their team's typical workflow as a combination of static analysis and fuzzing, stating, " Our bread and butter tends to be static analysis and fuzzing … using public rules and also writing our own internal rules. Often that informs the fuzzer." Likewise, P13 explained that "only when we combine [fuzzing and formal tools] do we find the bug," illustrating the need to augment fuzzing with complementary strategies to reach deeper system states.

As described above, newer hybrid fuzzing approaches attempt to integrate fuzzing with static analysis, but participants who had tried them found them difficult to use. In some cases, participants modified the fuzzer itself to accommodate their needs. P3, working with stateful protocols, recalled, "I had to make some modifications to [the fuzzer] … We used another state machine alongside [the fuzzer] to ensure certain states were being invoked or not," showing that integration sometimes required invasive changes to the tool.

For some participants (4/18), integrating fuzzing into their workflows required adaptations beyond configuration tweaks, including building their own wrappers, scripts, or orchestration layers. For example, P12 and P16 had used Cargo-Fuzz (a libFuzzer wrapper) to fuzz Rust programs, while P15 mentioned using TestFuzz and CargoAFL (an AFL wrapper) for Rust programs. P18 reported using GoFuzz for Go projects and noted limited success integrating fuzzing into a GitLab continuous integration (CI) pipeline. Others, like P6 and P7, wrote custom scripts to manage parallel runs or automate tasks like resetting parameters in order to scale fuzzing across multiple targets or campaigns. While effective, these workarounds highlight the absence of built-in support for usability and integration.

These adaptations highlight the diverse ways experienced users shape fuzzing workflows to fit real-world testing challenges: often blending fuzzers with other tools, tuning their behavior, or modifying them directly to meet their needs.

Suggestion: Fuzzers should be flexible and modular. In response to the growing complexity of fuzzers, participants advocated for flexibility in fuzzer structure and configuration. Some (4/18) expressed a preference for modular architecture rather than huge, monolithic tools; in a modular architecture, components (e.g., mutator, scheduler, and feedback mechanism) could be swapped or extended independently As P7 explained: "The program being executed, the mutation module, the scheduling module—they should all be replaceable, like plug-and-play components. … In [one tool]… if I want to change how feedback or scheduling works, I have to go into the source code and make changes directly. And that's not pretty."

[2]Similar concerns are reported in [27].

Others (6/18) expressed a preference for standalone tools or custom integrations over bundled toolchains. Rather than using pre-integrated modules or instrumentation layers, they want to combine tools manually to suit their specific goals. P4 said, "Some people might disagree with me, but I prefer standalone tools with their own setups."

Beyond architectural modularity, participants also emphasized the need for greater configuration flexibility. While many tools offer various flags and options, some participants feel that important parameters are too deeply embedded or hard-coded, limiting the ability to adapt the tool to different use cases. P9, for example, expressed frustration with one tool: "There are a lot of flags, but I wish there were more options to turn different things on and off. … A lot of things are baked in and hard-coded into the codebase, and I wish I could modify them." This lack of configurability makes it harder for participants to tailor fuzzers to fit specific performance, instrumentation, or deployment needs.

These suggestions show a broader desire for fuzzers that not only work out of the box, but also allow power users to adapt and reconfigure them for advanced use cases without digging deeply into the source code.

Suggestion: Increase applicability across problem domains. Many participants (11/18) further expressed a desire for fuzzers that could work across a wider range of domains in general. They feel that current tools are limited to traditional bug classes and struggle to scale to more complex or abstract targets, such as business logic, system integration, or semantic validation.

Some participants (3/18) suggested fuzzers should operate at a higher level of abstraction to better support application logic, such as authentication, authorization, or backend workflows. They envisioned tools that could combine internal application data (e.g., web server logs or state transitions) with fuzzer feedback to enable this shift. As P12 explained, "If you can reconcile what the web server sees with what the fuzzer sees, then I think there is opportunity."

Others (5/18) emphasized that certain domains, such as kernel fuzzing, emulator-based analysis, and distributed systems, remain prohibitively slow or difficult to fuzz effectively. They called for tools that could handle these environments more efficiently, especially when speed and depth must be balanced. As P14 noted, "General-purpose fuzzing is already fast, but kernel fuzzing, network fuzzing, and emulator-based fuzzing tend to be slow."

Participants (10/18) also suggested combining fuzzing with formal methods or symbolic techniques could improve its viability in domains with large design spaces or complex constraints. P13 said, "Fuzzing is more efficient if I have to explore a large design space, especially when combined with formal verifications." Overall, participants noted much potential value in hybrid approaches, despite the usability burdens they currently impose (discussed above), further reinforcing the need for more usable tooling in this space.

5 Discussion

Our findings demonstrate that limited fuzzer usability is not a minor inconvenience but a significant barrier with tangible security consequences. The steep learning curve forces practitioners into self-directed, trial-and-error learning, producing fragmented or unaligned mental models of how fuzzers operate. These misaligned models, combined with the high manual effort required for triage, prevent even enthusiastic and experienced users from realizing the full potential of fuzzing. Many of these issues stem from a fundamental mismatch between users' mental models and tools' complex, opaque reality. While prior work has emphasized setup hurdles for novice users [49, 50], our interviews with experienced practitioners additionally reveal deeper, persistent usability gaps across the fuzzing lifecycle, particularly the tension between novices' need for guidance and experts' demand for flexibility.

Some modern fuzzing frameworks, such as LibAFL's [15] modular architecture and OSS-Fuzz's [55] CI integration, have begun addressing aspects of these problems. However, these remain exceptions rather than the norm. Our recommendations therefore reflect broader usability gaps across the fuzzing ecosystem, grounded in the perspectives of users working across diverse environments and domains.

From guided setup to expert customization. A recurring challenge is the "one-size-fits-all" design of many fuzzers. For novices, prior studies highlight how steep setup curves deter adoption [49, 50]. Our findings reinforce this but extend it: even experienced users revert to trial and error due to inadequate defaults and poor documentation. This points to a clear need for fuzzers that function as adaptive guides, lowering barriers to entry with automated harness generation, smart defaults, and context-aware configuration assistance. These features would shift onboarding from frustrating trial and error to a more systematic process.

For some experts, however, the priorities differ. They stressed the importance of modular, API-driven architectures that allow core components like mutators, feedback mechanisms, and schedulers to be swapped or extended without editing source code. While recent frameworks like LibAFL move toward such modularity, this adaptability remains rare across the ecosystem. Supporting both guided setup and expert customization is crucial if fuzzers are to scale beyond niche adoption.

Making fuzzer output actionable. Output triage emerged as another significant bottleneck. Our study highlights practitioners' lack of trust in existing deduplication and categorization features, leading to heavy reliance on custom scripts. This makes raw output labor-intensive to transform into actionable insights.

Our participants also reported difficulty reasoning about fuzzer progress. Participants described fuzzers as "black boxes," offering opaque signals that make it difficult to judge coverage growth, seed quality, or when to stop. Participants consistently highlighted stopping criteria as a central pain point. Building richer runtime transparency would empower users to make informed stopping decisions and reduce wasted computation.

Workflow integration and interface improvements. Beyond output transparency, participants emphasized fuzzers' poor fit with existing workflows. Participants called for native integration with development ecosystems, including IDE plugins, dashboards, and version control hooks, allowing fuzzers to be run incrementally and monitored naturally.

These findings echo broader themes in security tool usability research, where mismatches with developers' workflows have driven distrust and abandonment [46]. For fuzzing to succeed as a practical testing strategy, outputs and operation must be tightly embedded into the developer workflow rather than presented as a standalone artifact.

Usability in emerging domains. While participants largely work with traditional grey-box fuzzers, they highlighted that hybrid fuzzing and other advanced approaches introduce even more usability hurdles. Research has emphasized the technical performance benefits of new approaches [26, 75], but their complexity has made adoption difficult in practice. Understanding the usability challenges of hybrid tools is an underexplored but critical research area.

Participants also imagined human-in-the-loop fuzzing systems where users could guide exploration, prioritize paths, or modify strategies mid-campaign, supported by intelligent recommendations and stopping criteria. These ideas resonate with trends toward collaborative and interpretable security tooling.

As fuzzing expands into hybrid analysis, CI/CD pipelines, and hardware or kernel testing, the principles highlighted here—guided setup, expert modularity, actionable feedback, and collaborative interaction—will become even more critical. Our work underscores that usability gaps persist beyond setup challenges, including for experienced practitioners, with systemic consequences for real-world adoption. Addressing these challenges is essential if fuzzing is to fulfill its promise not just as a technically powerful approach, but as a practically usable one.

6 Conclusion

Fuzzing continues to evolve as a powerful testing techniques, but its usability remains a critical barrier to broader adoption. Through 18 semi-strictured interviews with experienced users from both academia and industry, we identify recurring challenges in the usage of fuzzers across setup, feedback interpretation, output triage, and workflow integrations. Our findings highlight a disconnect between the growing technical capabilities of fuzzers and the practical needs of fuzzers users. By centering usability in future tool development through better automation, flexibility, actionable feedback, and integration, fuzzing as a testing technique can be more accessible and effective across a wider range of domains.

Acknowledgment. We gratefully acknowledge support from a UMIACS contract under the partnership between the University of Maryland and the Department of Defense. This study is supported in part by the National Science Foundation under grant number 2247954. We also thank all participants who generously took part in our study.

References

[1] Google 2025. *Google/Fuzzbench*. Google. https://github.com/google/fuzzbench
[2] Domagoj Babić, Stefan Bucur, Yaohui Chen, Franjo Ivančić, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. 2019. Fudge: Fuzz Driver Generation at Scale. In *ESEC/FSE 2019*. 975–985. doi:10.1145/3338906.3340456
[3] Antonia Bertolino. 2007. Software Testing Research: Achievements, Challenges, Dreams. In *FOSE 2007*. 85–103. doi:10.1109/FOSE.2007.25
[4] Paul E Black, Barbara Guttman, and Vadim Okun. 2021. Guidelines on Minimum Standards for Developer Verification of Software. arXiv:2107.12850
[5] Marcel Boehme, Cristian Cadar, and Abhik ROYCHOUDHURY. 2021. Fuzzing: Challenges and Reflections. 38, 3 (2021), 79–86. doi:10.1109/MS.2020.3016773
[6] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. 2020. Fuzzing: Challenges and Reflections. *IEEE Software* 38, 3 (2020), 79–86.
[7] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-Adaptive Mutational Fuzzing. In *S&P 2015*. IEEE, 725–741. doi:10.1109/SP.2015.50
[8] Maria Christakis and Christian Bird. 2016. What Developers Want and Need From Program Analysis: An Empirical Study. In *ASE 2016*. 332–343. doi:10.1145/2970276.2970347
[9] Victoria Clarke and Virginia Braun. 2017. Thematic Analysis. *The Journal of Positive Psychology* 12, 3 (2017), 297–298.
[10] Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang. 2023. Large language Models are Edge-Case Fuzzers: Testing Deep Learning Libraries via Fuzzgpt. arXiv:2304.02014
[11] 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. doi:10.1109/TSE.2020.3004525
[12] Max Eisele, Marcello Maugeri, Rachna Shriwas, Christopher Huth, and Giampaolo Bella. 2022. Embedded Fuzzing: A Review of Challenges, Tools, and Solutions. 5, 1 (2022), 18. doi:10.1186/s42400-022-00123-y
[13] Michael Felderer, Matthias Büchler, Martin Johns, Achim D Brucker, Ruth Breu, and Alexander Pretschner. 2016. Security Testing: A Survey. In *Advances in Computers*. Vol. 101. Elsevier, 1–51.
[14] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *WOOT 2020*. https://www.usenix.org/conference/woot20/presentation/fioraldi
[15] Andrea Fioraldi, Dominik Maier, Dongjia Zhang, and Davide Balzarotti. 2022. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *CCS 2022* (Los Angeles, U.S.A.). doi:10.1145/3548606.3560602
[16] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. 2008. Grammar-Based Whitebox Fuzzing. In *PLDI 2008*. 206–215. doi:10.1145/1375581.1375607
[17] Google. [n. d.]. *Google FuzzTest*. https://github.com/google/fuzztest
[18] Zhaoqiang Guo, Tingting Tan, Shiran Liu, Xutong Liu, Wei Lai, Yibiao Yang, Yanhui Li, Lin Chen, Wei Dong, and Yuming Zhou. 2023. Mitigating False Positive Static Analysis Warnings: Progress, Challenges, and Opportunities. *IEEE Transactions on Software Engineering* 49, 12 (2023), 5154–5188.
[19] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. In *POMACS 2020*, Vol. 4. 1–29. doi:10.48550/arXiv.2009.01120
[20] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L Hosking. 2021. Seed Selection for Successful Fuzzing. In *ISSTA 2021*. 230–243. https://hexhive.epfl.ch/publications/files/21ISSTA2.pdf
[21] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. FuzzGen: Automatic Fuzzer Generation. In *USENIX Security 2020*. 2271–2287.
[22] Bokdeuk Jeong, Joonun Jang, Hayoon Yi, Jiin Moon, Junsik Kim, Intae Jeon, Taesoo Kim, WooChul Shim, and Yong Ho Hwang. 2023. Utopia: Automatic Generation of Fuzz Driver Using Unit Tests. In *S&P 2023*. IEEE, 2676–2692. doi:10.1109/SP46215.2023.10179394
[23] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why Don't Software Developers Use Static Analysis Tools to Find Bugs?. In *ICSE 2013*. IEEE, 672–681. doi:10.1109/ICSE.2013.6606613
[24] Samantha Katcher, James Mattei, Jared Chandler, and Daniel Votipka. 2025. An Investigation of Interaction and Information Needs for Protocol Reverse Engineering Automation. In *CHI 2025*. doi:10.1145/3706598.3713630
[25] Natalia Kazankova. 2024. *From DAST to Dawn: Why Fuzzing is the Better Solution*. https://www.code-intelligence.com/blog/from-dast-to-dawn-why-fuzzing-is-the-better-solution
[26] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel.. In *NDSS*. https://www.ndss-symposium.org/wp-content/uploads/2020/02/24018-paper.pdf
[27] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *CCS 2018* (Toronto, Canada). NY, USA, 2123–2138. doi:10.1145/3243734.3243804
[28] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage. In *Proceedings of the 33rd ACM/IEEE Int'l Conf. on automated software engineering*. 475–485. doi:10.1145/3238147.3238176
[29] Jinfeng Li. 2020. Vulnerabilities Mapping Based on OWASP-SANS: A Survey for Static Application Security Testing (SAST). arXiv:2004.03216
[30] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the Art. *IEEE Transactions on Reliability* 67, 3 (2018), 1199–1218.
[31] Bingchang Liu, Liang Shi, Zhuhua Cai, and Min Li. 2012. Software Vulnerability Discovery Techniques: A Survey. In *2012 Fourth Int'l Conf. on Multimedia Information Networking and Security* (2012-11). 152–156. doi:10.1109/MINES.2012.202

[32] Yuwei Liu, Yanhao Wang, Purui Su, Yuanping Yu, and Xiangkun Jia. 2021. Instruguard: Find and Fix Instrumentation Errors for Coverage-Based Greybox Fuzzing. In *ASE 2021*. 568–580. https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=9678671

[33] LLVM Project. [n. d.]. *LibFuzzer – A Library for Coverage-Guided Fuzz Testing. — LLVM 21.0.0git Documentation*. https://llvm.org/docs/LibFuzzer.html

[34] Chenyang Lyu, Shouling Ji, Yuwei Li, Junfeng Zhou, Jianhai Chen, and Jing Chen. 2018. Smartseed: Smart Seed Generation for Efficient Fuzzing. arXiv:1807.02606

[35] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *USENIX Security 2019*. 1949–1966. doi:10.5555/3361338.3361473

[36] Chenyang Lyu, Shouling Ji, Xuhong Zhang, Hong Liang, Binbin Zhao, Kangjie Lu, and Raheem Beyah. 2022. EMS: History-Driven Mutation for Coverage-Based Fuzzing. In *NDSS*. https://www.ndss-symposium.org/wp-content/uploads/2022-162-paper.pdf

[37] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. 2024. Prompt Fuzzing for Fuzz Driver Generation. In *CCS 2024*. 3793–3807. doi:10.48550/arXiv.2312.17677

[38] Reza M. Parizi, Kai Qian, Hossain Shahriar, Fan Wu, and Lixin Tao. 2018. Benchmark Requirements for Assessing Software Security Vulnerability Testing Tools. In *COMPSAC 2018*, Vol. 01. 825–826. doi:10.1109/COMPSAC.2018.00139

[39] Sanoop Mallissery and Yu-Sung Wu. 2023. Demystify the Fuzzing Methods: A Comprehensive Survey. *Comput. Surveys* 56, 3 (2023), 1–38.

[40] James Mattei, Madeline McLaughlin, Samantha Katcher, and Daniel Votipka. 2022. A Qualitative Evaluation of Reverse Engineering Tool Usability. In *ACSAC 2022* (Austin, TX, USA). NY, USA, 619–631. doi:10.1145/3564625.3567993

[41] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. 2019. Reliability and Inter-Rater Reliability in Qualitative Research: Norms and Guidelines for CSCW and HCI Practice. In *CSCW*, Vol. 3. 1–23. doi:10.1145/3359174

[42] Richard McNally, Ken Yiu, Duncan Grove, and Damien Gerhardy. 2012. *Fuzzing: The State of the Art*. Technical Report DSTO–TN–1043. Australian Government, Department of Defence, Defence Science and Technology Organisation.

[43] Marcus Nachtigall, Michael Schlichtig, and Eric Bodden. 2022. A Large-Scale Study of Usability Criteria Addressed by Static Analysis Tools. In *ISSTA 2022*. 532–543. doi:10.1145/3533767.3534374

[44] Timothy Nosco, Jared Ziegler, Zechariah Clark, Davy Marrero, Todd Finkler, Andrew Barbarello, and W. Michael Petullo. 2020. The Industrial Age of Hacking. In *USENIX Security 2020*. 1129–1146. https://www.usenix.org/conference/usenixsecurity20/presentation/nosco

[45] 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 Trans. Softw. Eng. Methodol.* 33, 1, Article 14 (Nov. 2023), 26 pages. doi:10.1145/3611668

[46] 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.

[47] OpenAI. 2025. OpenAI/Whisper. https://github.com/openai/whisper. https://github.com/openai/whisper

[48] OSS-Fuzz. [n. d.]. *Fuzzing Introspection of OSS-Fuzz Projects*. https://introspector.oss-fuzz.com/

[49] 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 *SOUPS 2021*. 553–572. https://www.usenix.org/conference/soups2021/presentation/ploger

[50] Stephan Plöger, Mischa Meier, and Matthew Smith. 2023. A Usability Evaluation of AFL and libFuzzer with CS Students. In *CHI 2023* (Hamburg, Germany). NY, USA, Article 186, 18 pages. doi:10.1145/3544548.3581178

[51] Roshan Namal Rajapakse, Mansooreh Zahedi, and Muhammad Ali Babar. 2021. An Empirical Analysis of Practitioners' Perspectives on Security Tool Integration Into DevOps. In *ESEM 2021*. 1–12.

[52] Thorsten Rangnau, Remco v. Buijtenen, Frank Fransen, and Fatih Turkmen. 2020. Continuous Security Testing: A Case Study on Integrating Dynamic Security Testing Tools in CI/CD Pipelines. In *EDOC 2020*. IEEE, 145–154. doi:10.1109/EDOC49727.2020.00026

[53] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing Seed Selection for Fuzzing. In *USENIX Security 2014*. 861–875.

[54] Irving Seidman. 2006. *Interviewing as Qualitative Research: A Guide for Researchers in Education and the Social Sciences*. Teachers college press.

[55] Kostya Serebryany. 2017. OSS-Fuzz - Google's Continuous Fuzzing Service for Open Source Software. In *USENIX security 2017*. USENIX Association, Vancouver, BC.

[56] Dongdong She, Abhishek Shah, and Suman Jana. 2022. Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis. In *S&P 2022*. IEEE, 2194–2211. doi:10.48550/arXiv.2203.12064

[57] Gabriel Sherman and Stefan Nagy. 2025. No Harness, No Problem: Oracle-Guided Harnessing for Auto-generating C API Fuzzing Harnesses. In *ICSE 2025*. IEEE Computer Society, 775–775. https://www-old.cs.utah.edu/~snagy/papers/25ICSE-b.pdf

[58] Justin Smith, Lisa Nguyen Quang Do, and Emerson Murphy-Hill. 2020. Why Can't Johnny Fix Vulnerabilities: A Usability Evaluation of Static Analysis Tools for Security. In *SOUPS 2020*. 221–238. doi:10.5555/3488905.3488918

[59] 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 *ESEC/FSE 2015*. 248–259. doi:10.1145/2786805.2786812

[60] Murugiah Souppaya, Karen Scarfone, and Donna Dodson. 2022. Secure Software Development Framework. *NIST Special Publication* 800, 218 (2022), 800–218.

[61] Mohammad Tahaei, Ruba Abu-Salma, and Awais Rashid. 2023. Stuck in the Permissions With You: Developer & End-User Perspectives on App Permissions & Their Privacy Ramifications. In *CHI 2023*. 1–24. doi:10.48550/arXiv.2301.06534

[62] 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 *CHI 2021*. 1–17. doi:10.1145/3411764.3445616

[63] 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 *SOUPS 2016*. https://www.usenix.org/system/files/conference/soups2015/wsiw16_paper_thomas.pdf

[64] Omer Tripp, Salvatore Guarnieri, Marco Pistoia, and Aleksandr Aravkin. 2014. Aletheia: Improving the Usability of Static Security Analysis. In *CCS 2014*. 762–774. doi:10.1145/2660267.2660339

[65] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C Gall, and Andy Zaidman. 2020. How Developers Engage with Static Analysis Tools in Different Contexts. *Empirical Software Engineering* 25 (2020), 1419–1457.

[66] 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 *SANER 2018*. 38–49. doi:10.1109/SANER.2018.8330195

[67] Daniel Votipka, Rock Stevens, Elissa Redmiles, Jeremy Hu, and Michelle Mazurek. 2018. Hackers vs. Testers: A Comparison of Software Vulnerability Discovery Processes. In *2018 IEEE Symp. on Security and Privacy (SP)*. 374–391. doi:10.1109/SP.2018.00003

[68] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *2017 S&P*. IEEE, 579–594. doi:10.1109/SP.2017.23

[69] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-Aware greybox fuzzing. In *ICSE 2019*. IEEE, 724–735. doi:10.1109/ICSE.2019.00081

[70] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. 2020. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization.. In *NDSS 2020*. https://www.ndss-symposium.org/wp-content/uploads/2020/02/24422-paper.pdf

[71] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2024. Fuzz4all: Universal Fuzzing with Large Language Models. In *ICSE 2024*. 1–13. doi:10.48550/arXiv.2308.04748

[72] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. 2016. Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study. In *S&P 2016*. 158–177. doi:10.1109/SP.2016.18

[73] Qian Yan, Minhuan Huang, and Huayang Cao. 2022. A Survey of Human-Machine Collaboration in Fuzzing. In *DSC 2022*. IEEE, 375–382. doi:10.1109/DSC55868.2022.00058

[74] Zhenhua Yu, Zhengqi Liu, Xuya Cong, Xiaobo Li, and Li Yin. 2024. Fuzzing: Progress, Challenges, and Perspectives. 78, 1 (2024), 1–29. doi:10.32604/cmc.2023.042361

[75] 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 2018*. 745–761. doi:10.5555/3277203.3277260

[76] Cen Zhang, Mingqiang Bai, Yaowen Zheng, Yeting Li, Wei Ma, Xiaofei Xie, Yuekang Li, Limin Sun, and Yang Liu. 2023. Understanding Large Language Model Based Fuzz Driver Generation. arXiv:2307.12469

[77] Cen Zhang, Xingwei Lin, Yuekang Li, Yinxing Xue, Jundong Xie, Hongxu Chen, Xinlei Ying, Jiashui Wang, and Yang Liu. 2021. APICraft: Fuzz Driver Generation for Closed-Source SDK Libraries. In *USENIX Security 2021*. 2811–2828. https://www.usenix.org/conference/usenixsecurity21/presentation/zhang-cen

[78] Cen Zhang, Yaowen Zheng, Mingqiang Bai, Yeting Li, Wei Ma, Xiaofei Xie, Yuekang Li, Limin Sun, and Yang Liu. 2024. How Effective are They? Exploring Large Language Model Based Fuzz Driver Generation. In *ISSTA 2024*. 1223–1235. doi:10.48550/arXiv.2312.17677

[79] Mingrui Zhang, Jianzhong Liu, Fuchen Ma, Huafeng Zhang, and Yu Jiang. 2021. Intelligen: Automatic Driver Synthesis for Fuzz Testing. In *ICSE 2021*. IEEE, 318–327. doi:10.1109/ICSE-SEIP52600.2021.00041

[80] Xiaoqi Zhao, Haipeng Qu, Wenjie Lv, Shuo Li, and Jianliang Xu. 2021. Moofuzz: Many-Objective Optimization Seed Schedule for Fuzzer. *Mathematics* 9, 3 (2021), 205.

[81] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A Survey for Roadmap. *Comput. Surveys* 54, 11s (2022), 1–36.

A Appendix: Interview Protocol

Our codebook, pre-interview survey, and interview protocol are publicly available at our OSF repository: https://osf.io/bshup/?view_only=078ea6a674cb4044b23334554654880a.