A Qualitative Analysis of Fuzzer Usability and Challenges

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

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

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

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

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

Michelle Mazurek
University of Maryland
College Park, 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. The 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.

ACM Reference Format:
Yunze Zhao, Wentao Guo, Harrison Goldstein, Daniel Votipka, Kelsey Fulton, and Michelle Mazurek. 2018. A Qualitative Analysis of Fuzzer Usability and Challenges. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Submitted to CCS).* ACM, New York, NY, USA, 16 pages. https://doi.org/XXXXXXX.XXXXXXX

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 [15, 42, 51, 65, 78].

Fuzzing's efficacy for vulnerability discovery is well-documented, with tools 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 [1, 2, 8, 14, 16]. 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 [7, 26, 58].

The majority of fuzzing research has, unsurprisingly, prioritized technical advancements, such as input generation [34, 66], seeds scheduling [54, 77], mutation [10, 29], and harness generation [5, 21, 55], with relatively little attention to how users interact with and adapt these tools in practice.

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 [6, 45, 48]. Grey-box fuzzers like AFL and libFuzzer rely on code coverage feedback to guide testing and discover bugs, but the configuration and tuning of these tools for optimal performance demand a deep understanding of the underlying mechanisms [8, 16, 78]. Challenges with configuration and associated workflows can lead to frustration and misconfiguration, reducing the potential impact of fuzzing in everyday software development practices [32, 78]. Further, a lack of standardized evaluation practices makes it difficult for adopters of fuzzers to compare options based on performance [19, 28]

These prior studies have made an important start toward understanding fuzzing usability, but 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 fuzzing tools would better support practical adoption and real-world workflows?

Our findings reveal that while participants viewed fuzzing as powerful and indispensable, they frequently encountered steep learning curves, unintuitive interfaces, and limited feedback during both setup and analysis. Participants often relied on informal heuristics and self-directed learning, leading to inconsistent mental models of how fuzzers operate and creating confusion. 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 its fundamental concepts, popular techniques, and widely used tools. Then we discuss studies of fuzzers' usability and the usability 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 the target with a wide range of inputs, then observing runtime outcomes. A fundamental assumption in fuzzing is that reliability or 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 the 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 [17, 31, 42, 70, 71, 78]. Coverage-guided grey-box fuzzing tools (e.g., AFL, AFL++, libFuzzer, and Honggfuzz) dominate both research and practice due to their balance between effectiveness and scalability [16, 78]. We primarily recruited participants with 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 [31, 42, 70, 71, 78], 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 instrumented to enable runtime feedback [68]. 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 [33].

Input delivery: 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 [4], expect users to supply a test harness, as described above. The harness must be 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 [16]. These tools inject inputs via stdin or temporary files and 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 [5, 22, 74, 76], sometimes involving large language models [37, 73, 75]. These approaches, while promising, 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 fuzzer generates inputs to explore different execution paths [31]. Most modern fuzzers come with optimized, built-in mutation strategies that can be effective out of the box [20, 29, 35, 36, 52]. Instead of adjusting the mutation strategy, users typically provide example seed inputs, enable grammar support [67], 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, 52].

Monitoring and interpreting program behavior. In coverage-guided fuzzing, users track dynamic feedback such as code coverage, execution counts, or branch exploration to assess if the fuzzer is exercising new paths, and 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. The fuzzer reports 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, there has been some work comparing various tools and contexts of use in controlled experiments [44, 48, 49]. For example, Nosco et al. conducted an experiment with 12 security professionals aimed at comparing the efficacy of breath- vs. depth-first review [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 [48]. 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 tool instrumentation, results interpretation, and writing harnesses [49]. 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, in many cases participants' lack of experience prevented them from reaching later stages of the fuzzing process, limiting the challenges identified to fuzzer setup. By interviewing experts with years of fuzzing experience testing multiple programs, we are able to identify a broader range of challenges, as well as strategies employed by our participants to overcome these barriers

The second category of work identifies fuzzing challenges writ broadly through surveys, expert discussion, and artifact analysis [9, 45]. Böhme et al. synthesized discussions from a working group of 31 fuzzing practitioners and researchers, identifying multiple fuzzing challenges [9]. They then surveyed 21 leaders in industry and academia outside the working group to confirm this challenge list. 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 it with a survey of 103 developers who indicated how often they experienced each [45]. Their taxonomy identifies 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 to 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 [13, 18, 30, 56, 60, 61, 63], lack clear outputs [13, 38, 43, 56, 57, 59], or be hard to configure [43, 56, 59, 62] and integrate into existing workflows [11, 24, 50]. Rangnau et al. studied dynamic testing beyond fuzzing, focusing on its integration into developers' workflows [51]. Mattei et al. performed a heuristic evaluation of 288 security tools (including static and dynamic analysis tools, including fuzzers), assessing their expected usability [40]. This review, which drew on prior interviews with security profesionals [64], 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 design recommendations for protocol reverse engineering human-automation interfaces [25]. Yakdan et al. developed a decompiler designed for code readability and showed that it improved program understanding [69]. 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 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, potential participants completed an initial survey that collected demographic and contextual information, including participants' 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. While many fuzzers exist, we primarily focused on users of mainstream industry tools (AFL, libFuzzer, and LibAFL), including potential participants who built custom wrappers around these tools [39]. This approach helped us capture a more comprehensive picture of the challenges and strengths of commonly used fuzzers. As recruitment proceeded, we employed purposive sampling, i.e., selecting new participants to increase the diversity of fuzzing goals and experiences within our sample.

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

- **Security Professionals**: 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 Researchers**: We contacted authors of recent research papers that involved applying fuzzers. Our goal was to recruit researchers with hands-on experience using fuzzing in

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

varied contexts, while excluding those whose primary experience was limited to developing a fuzzer.

- **Community Announcements**: We posted recruitment messages in multiple fuzzing-related Discord servers. These communities included practitioners, researchers, and hobbyists, helping us reach a broader pool of participants.
- **Snowball Sampling and Professional Connections**: We recruited additional participants through personal networks and referrals, including asking initial 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 Anonymous Institution's Institutional Review Board (IRB), all participants were informed about the purpose and scope of the study prior to participation. Participants were shown a consent form at the beginning of the prescreening 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 under participants' permissions. Recordings were stored locally on secure, access-controlled machines. Identifiable information was removed or anonymized during transcriptions and analysis. Although some participants discussed professional or project-related tooling, we did not include proprietary or sensitive technical details in our findings.

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

- **Fuzzer Setup and Usage:** Participants described how they selected and prepared 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:** Participants reflected on potential enhancements to fuzzing tools, 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 prescreening 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.

3.3 Data Analysis

All interviews were transcribed using OpenAI Whisper [47], an automated transcription service. The model was run entirely on local machines. To ensure data security and participant privacy, all audio recordings and transcripts were stored locally on secure systems and were not shared with third-party services. The transcripts were analyzed using a collaborative thematic analysis to identify recurring themes and insights [12], through the following stages:

1. **Initial Open Coding:** Two researchers independently reviewed the transcripts and assigned preliminary codes based on recurring themes observed in the data. This step ensured a broad and unbiased capture of potential insights.
2. **Collaborative Code Refinement:** The two researchers compared and discussed their initial codes, collaboratively resolving any discrepancies and agreeing on a unified set of codes. This process ensured consistency and rigor in the coding framework.
3. **Final Codebook Development:** The agreed-upon codes were refined into a final codebook, with categories structured to align with the study's research questions. This included themes for workflows, usability challenges, and improvement opportunities.

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 (IRR), 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 the Section 4.

3.4 Threats to validity

While our study included participants with a range of fuzzing experience, we focused on widely used tools such as AFL/AFL++, libFuzzer, and their variants. These tools are the most common entry points into fuzzing workflows and are frequently cited in both academic research and practical security engineering. As such, we aimed to capture usability patterns and challenges most representative of current practice, at the potential cost of underrepresenting experiences with niche fuzzers or highly specialized domains. Accordingly, our interview protocol emphasized conceptual and methodological themes that transcend specific tooling.

Another limitation is the potential self-selection bias. Participants who responded to our outreach likely have stronger interest in fuzzing and improving fuzzing workflows, and may therefore have more interest or experience in overcoming tooling challenges. We may therefore miss some challenges experienced by those who tried but quickly 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 [53].

We did not explicitly measure saturation during the interview process. However, as the study progressed, we observed convergence in participants' responses, suggesting that our results capture core usability challenges and opportunities for improvement.

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 6 categories. We begin with participant demographics to contextualize their experiences. Next, we describe participants' mental models of fuzzers. We then present findings (challenges, practices, and suggestions for improvement) across three stages of the fuzzing workflow: (1) preparing fuzzing targets and configuring fuzzers, (2) running and monitoring fuzzing campaigns, and (3) integrating and extending fuzzing within broader development and testing workflows.

4.1 Demographics

We categorized participants' use cases for fuzzers based on their roles and goals. Participants often engaged in multiple use cases, which we captured across the following major categories:

- Software development: Using fuzzers as part of regular development workflows to catch "low hanging fruits" bug 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 industrial 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. The distribution of participant experience, tool usage, and organizational context is summarized in Table 1.

4.2 Participants' understanding of fuzzers

While all our study participants were experienced fuzzer users, and many had deep technical knowledge, including research backgrounds, their understanding of how fuzzers work varied widely. Some participants had highly developed understandings of fuzzing internals, while others expressed uncertainty about how fuzzers work "under the hood." This variation often reflected 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 as a foundational concept in their approach to understanding software behavior. Even when fuzzers did not fully align with their workflows or goals, participants saw fuzzing as indispensable for surfacing bugs other methods might miss. Rather than viewing it as just another automation tool, many regarded 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 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, particiapants 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 considered fuzzing indispensable, participants recognized 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 were widely seen as well suited to fuzzing compared to software with complex state or semantic logic.

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

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

Fuzzers are black boxes causing 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 understood how fuzzers *should* work they found that real-world use often offered little feedback about what was happening 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 relied on surface-level metrics such as line or path coverage, crash counts, or timeouts to gauge progress. While they recognized these metrics were imperfect, they were 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 contributed 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 faced preparing fuzzing targets and configuring fuzzers, and how these challenges impacted their ability to apply fuzzing effectively.

ID	Role	Exp.	Tools	Org. Type	Fuzzing Tool Use Cases				
					Software Dev	Cybersecurity	DevOps	QA /Testing	Research
P1	Software Engineer	1-2 yrs	●, △, ★	Large	✓			✓	✓
P2	Research Assistant	1-2 yrs	●, △	Academia		✓			✓
P3	Research Assistant	1-2 yrs	●	Academia					✓
P4	Software Engineer	>5 yrs	●, △, ★, ◊	Large	✓	✓			✓
P5	Research Assistant	3-5 yrs	●, △	Academia		✓			✓
P6	Research Assistant	1-2 yrs	●, △, ∇, ◊, ★	Academia	✓	✓	✓		✓
P7	Security Engineer	>5 yrs	●, △, ★	Large	✓				✓
P8	Security Researcher	<1 yr	●, ∇	Academia	✓		✓	✓	
P9	Research Assistant	1-2 yrs	●, △, ∇	Academia	✓			✓	✓
P10	Research Assistant	3-5 yrs	●, ★	Academia		✓			✓
P11	Software Engineer	>5 yrs	●, △, ∇	Large	✓	✓			
P12	Security Engineer	3-5 yrs	●, △, ★	Small/Med	✓	✓		✓	
P13	Research Assistant	3-5 yrs	●, ◊	Academia	✓				✓
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	●, △, ★	Academia	✓				✓
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; Academia = university/industrial research.

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 labels **Challenge**, **Practice**, and **Suggestion** to clearly signal the focus of each finding.

Challenge: Self-directed learning leads to inconsistent understanding and conceptual ambiguity. A root cause of many challenges is how participants learned to use fuzzers. Most participants described learning fuzzing through hands-on practice (11/18) and/or written resources, e.g., blogs, academic papers, and documentation (13/18). While this approach often worked for quickly setting up fuzzers or understanding fuzzing at a high level, it lead to 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 did not provide sufficient guidance, some participants (6/18) reported turning to fuzzers' source codeto understand how specific components worked 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."

Participants noted that the absence of centralized, structured, or pedagogically designed learning resources—on top of fuzzing's already steep learning curve—made it difficult to develop a deep, coherent understanding of core concepts. Even technically advanced participants relied on ad-hoc sources and personal heuristics, contributing to inconsistent mental models between participants. This complicated on-boarding for new users and also introduced friction when sharing workflows, debugging issues collaboratively, or evaluating fuzzing effectiveness in a reproducible way.

P14 illustrated this challenge by reflecting on inconsistent practices even among experienced practitioners: "They often ask how to instrument a program properly because a lot of techniques require modifying the compiler… Each fuzzer has its own instrumentation approach." These tool-specific assumptions, left unexplained in documentation or on-boarding, often required participants to reverse engineer best practices on their own.

Because participants primarily learned by doing, their underlying understanding of fuzzing concepts was influenced strongly by the fuzzing contexts they had experienced, leading to divergent beliefs and assumptions. For example, participants exhibited varying interpretations of terms like "false positive," reflecting differences in fuzzing goals and contexts rather than simple confusion. P1 described false positives as duplicate crashes, putting focus on output triage and tooling efficiency. In contrast, P13 defined them as inputs that trigger unexpected behavior but do not correspond to actual vulnerabilities, suggesting a goal of surfacing security-critical flaws. Meanwhile, P17 considered false positives to be crashes caused by issues in the harness itself, reflecting a focus on validating the correctness of the fuzzing setup.

These differing perspectives highlights users learn fuzzing through the lens of their specific testing objectives. Without shared conceptual scaffolding, even experienced users may form divergent assumptions about tool behavior and goals, fragmenting community understanding.

Challenge: Setup is confusing and poorly aligned with participant needs. To help with setup, most fuzzers come with documentation, Docker containers, or tutorials. However, participants said these were 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." What's more, 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 the setup challenges, participants frequently encountered challenges preparing real-world programs for fuzzing, especially for large, layered, or legacy codebases. Many of these systems were poorly documented, difficult to build, or relied 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 lacked principled strategies for selecting where to fuzz, especially when they were fuzzing unfamiliar code. Instead, they relied 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 compounded participants' existing uncertainty about the fuzzer. Participants were 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 these setup frustrations, participants prioritized getting their chosen fuzzer up and running with minimal friction. Rather than aiming for a perfect setup from the start, they treated 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 valued speed and feedback over precision: they expected early runs to be shallow or error-prone, but saw value in getting something running quickly so they could 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 sought 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 also often started with minimal harnesses, often "feed[ing] the fuzzer input to whatever function in the API looks most top-level." (P16). These lightweight setups allowed them to validate that the tool was functioning before adding complexity.

Practice: Selecting fuzzing targets based on experience, intuition, and documentation. A critical part of setup involves identifying where to fuzz within a fuzzing target. As noted earlier, participants lacked principled strategies for target selection and often defaulted to intuition. Even when external tools were available to suggest candidate entry points, participants still shouldered 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 deeper understanding of an unfamiliar fuzzing target, participants used 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 documentationI can look at that."

When fuzzing workflows required or would benefit from initial seeds, most participants extracted them directly from the fuzzing target. A common strategy was to re-purpose unit test inputs or files from the project's test suite.P18 considered 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 wanted 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 P15 and P12 envisioned a possible VS Code plugin that could automate common setup steps: "a plugin that looks for targets, suggests a harness with an LLM, 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 wanted assistance with identifying good entry points and configuring fuzzer parameters. These tasks often relied on intuition or partial static analysis, but participants believed they 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 found it difficult to assess whether their configuration choices, such as seed selection or entry point targeting, were likely to yield useful results. Several expressed interest in tools that 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 fuzzing 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."

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, interpret output, and decide how long to continue. In this section, we describe the challenges participants encountered during active fuzzing, including difficulties interpreting fuzzer output and uncertainty about when to stop. We highlight the practical strategies participants used 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 did not expect crash reports to serve as a complete diagnosis, participants emphasized that outputs often lacked 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 required considerable post-processing and external analysis to become actionable.

Participants reported receiving large volumes of redundant crashes that required 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 de-duplication or grouping to minimize redundancy, participants lacked trust in these feature due to unreliability and a concern that unique bugs might be incorrectly grouped or silently discarded P11 said that despite AFL's built-in de-duplication, "We still found cases where different crashes actually stemmed from the same bug but were recorded as distinct." P7 noted the opposite problem: "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."

In addition to crash reports, fuzzers produce timeouts reports; participants also found these difficult to interpret. While many considered timeouts a potentially valuable signal—often indicating hangs or performance bottlenecks—fuzzers rarely provided 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 left participants uncertain whether to treat timeouts as bugs worth triaging or simply as noise.

Challenge: Runtime outputs lack key details and context. Beyond crash output, participants also described frustration with output produced while fuzzing is actively running. Many said it was difficult to tell whether their campaign was making progress. While some fuzzers exposed performance metrics like coverage growth or seed queue size, these metrics were 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 attempted to inspect logs during fuzzing found that crucial contextual information was 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 found 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. Even if a fuzzer discovers an issue once, it might fail to do so in the next repetition." Without detailed metadata about input provenance and mutation history, participants struggled 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 often generate a lot of output, but it is not informative. Participants had 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 understood that new bugs could always be found with more time or better mutations. However, real-world constraints like compute budgets or project deadlines meant they eventually had to decide when to stop. Making that decision, however, was far from straightforward.

Some participants (5/18) explicitly mentioned that they relied 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 tried to use metrics like branch or path coverage to guide the decision. But even this approach was fraught with doubt, especially when participants were unsure whether stagnation reflected 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 defaulted 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, adopted 24 hours as a standard campaign length, not because it was ideal, but because it aligned with common benchmarking practices.

While participants didn't expect fuzzers to offer a definitive "you're done" signal, they consistently expressed a desire for more meaningful feedback about progress. The broader challenge wasn't just about finding a stopping rule, but about dealing with the fundamental fuzzing uncertainty in more informed ways. As long as fuzzers remain opaque about how much ground they've covered users will continue to face ambiguity in balancing thoroughness with resource constraints.

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

Several participants (5/18) highlighted the importance of directly observing fuzzer behaviorduring 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 was especially important with limited tool support for runtime diagnostics and termination guidance. In practice, participants relied on a small set of interpretable signals, such as coverage, execution speed, to assess fuzzing progress.

Others turned to parallelism to maximize fuzzer efficiency and returns. Some ran multiple fuzzers or instances of the same fuzzer with synchronized corpora to increase path discovery. P14 combined 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 de-duplication, then manual triage. Participants followed a common pattern when analyzing fuzzing output: first de-duplicate crashes, then manually investigate those that remain. Many relied 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 was essential to reduce noise and focus attention on distinct issues.

After de-duplication, participants shifted to more in-depth analysis, such as re-running crashing inputs, tracing execution in tools like GDB or Valgrind, or inspecting code to identify root causes. While time-consuming, this manual process was necessary to understand the significance of each crash and determine whether it indicated a real vulnerability. As P17 explained, "In some cases, you can look it up online and try to see if other people experienced a similar crash."

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 deviated from tool-provided heuristics, reflecting the extent to which participants had to rely on personal judgment and scripting to make the output manageable. While these techniques made analysis more tractable, they were also labor intensive and required judgment and scripting skills.

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

Several participants (6/18) emphasized the need for richer, real-time feedback during fuzzing execution. Rather than waiting until the end of a run, they wanted visibility into the internal behavior of the fuzzer while it was running—such as which paths were being hit, how frequently, and where progress was 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 run-time 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 overlays 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 wanted 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 de-deduplication approaches left room for improvement, often requiring participants to develop custom solutions.

Beyond de-duplication, participants wanted 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 tools like GDB provide, 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 wanted 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 needed to adapt fuzzers to fit broader development workflows or support more advanced testing goals. In this section, we describe the limitations participants encountered when applying advanced features, customizing tool behavior, or scaling fuzzing across large systems. We outline the workarounds participants 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 found 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 introduced 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 were difficult to modify.

Several participants noted that without usable, transparent interfaces, new fuzzing tools that are theoretically better for a given task may not get adopted. They mentioned attempting and failing to use a specialized tool that was designed for their intended use case, as P3 explained "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 ran multiple analysis modes simultaneously, which made 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 found that tools didn't scale well and required 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 weren't just technical annoyances; they introduced inefficiencies that drained resources and limited 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 saw as overly focused on 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, or deep state exploration with a lack of attention to usability, such as clearer configuration interfaces, modularizing components, or improved crash triage.

P12 observed, "A lot of research is about maximizing coverage or depth, 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 considered 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]."

Across these ideas, the core message was consistent: "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." (P12) Participants don't just want better results or better performance, they want better visibility, smarter feedback, and tighter integration with their workflow.

Participants across both academia and industry emphasized that real-world adoption depends on usability. 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 [3], "there's no standardized way to evaluate…In this environment, a

[2]Similar concerns are reported in [28].

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 practical value.

Beyond research settings, lack of standardization also made collaboration and reproducibility more difficult. Differences in architecture, environments, or dependency versions created 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 selected and adapted tools based on 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 made pragmatic choices, such as targeting compiled binaries, specific APIs, or protocol-level behavior, based on what the context demanded. These decisions were 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 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 fuzzing with other tools, tuning its behavior, or modifying it 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 how fuzzer structure and configuration.Some (4/18) preferred a modular architecture rather than huge, monolithic tools, where 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."

Others (6/18) preferred standalone tools or custom integrations over bundled toolchains. Rather than using pre-integrated modules or instrumentation layers, they wanted 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 felt that important parameters were 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 made 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 felt that current tools were limited to traditional bug classes and struggled 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 found 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 results suggest the usability of fuzzers in practice remains a major barrier to wide adoption. Here, we discuss how insights from our study can inform the design of more usable fuzzers and outline opportunities for future work.

Some modern fuzzing frameworks (LibAFL's modular architecture, OSS-Fuzz's CI integration) have begun addressing some issues we highlight. However, our recommendations reflect broader usability gaps across the fuzzing ecosystem, from the perspective of users working in varied environments and domains.

Iterative setup with better defaults and automation. Our participants commonly approached fuzzing setup as a trial-and-error process: run the tool, observe failures, adjust the harness, and repeat. While this process can work, it tends to be time-consuming and labor intensive, especially for users who are not familiar with the fuzzing target. To better support getting started, fuzzers should provide better defaults, guided setup, and lightweight automation to help users get to a runnable state faster. For example, based on the characteristics of the fuzzing target or specific function of entry, tools could auto-generate basic harness scaffolding, recommend commonly used flags, or detect problematic configurations before launch. These improvements would lower the barrier to entry, reduce early-stage frustration, and allow users to focus their effort on campaign tuning rather than setup debugging.

Modular, flexible architectures for advanced users. While some participants preferred simple, ready-out-of-the-box setups, others—particularly those with security or research experience—sometimes preferred greater flexibility to tailor the fuzzer to their specific needs. These users expressed frustration with tool designs that made it difficult to replace or extend core components such as mutators, feedback mechanisms, or scheduling strategies. A modular architecture, where components can be swapped via well-defined APIs, would allow users to customize and experiment without modifying source code directly. For example, users should be able to plug in a custom mutation strategy or modify feedback logic without breaking tool stability. While frameworks like LibAFL have started moving in this direction, participants noted that this level of modularity is still uncommon across the broader fuzzing ecosystem.

Improved output triage and crash de-duplication. Output analysis currently requires manual inspection; however, participants typically wrote custom scripts to identify crashes worth investigating, because built-in de-duplication features were often insufficient. Tools should provide multiple crash de-duplication mechanisms that are more robust, transparent, and flexible, as well as offering summaries or filtering based on possible root cause. Participants also expressed interest in customizable crash categorization: using user-defined grouping logic rather than static, preset labels, to enable triage that matched their context and priorities.

Better workflow integration and interface improvements. Participants described fuzzers as rigid and disconnected from their broader workflows. They expressed a clear need for fuzzers to integrate more seamlessly into existing tooling (e.g., version control systems, build pipelines, and IDEs) so they can be deployed incrementally and monitored more naturally. In addition, participants wanted better user interaction. While participants had mixed attitudes toward graphical user interfaces overall, they all agreed that dashboards, lightweight visualizations, or IDE plugins could greatly improve visibility into what the fuzzer is doing, helping with campaign setup and monitoring. Others proposed interactive interfaces that could suggest possible harness improvements, forecast seed quality, or notify users when fuzzing has likely stalled.

5.1 Future work

While our participants largely worked with traditional fuzzers, a few had tried hybrid fuzzers. Despite their growing adoption in research and practice [72], hybrid fuzzers remain unexplored from a usability perspective, as existing studies tend to focus on technical performance and coverage improvements [23, 27, 46]. As hybrid fuzzing continues to evolve, understanding its usability implications remains an important but underexplored area.

Another direction is the development of human-in-the-loop fuzzing systems, where users can adjust input strategies, prioritize paths, or guide the fuzzer when progress stalls. Researchers could investigate what human input is most useful, how to design appropriate interfaces, and how to balance control with automation.

Future research should also explore how users make decisions about when to stop fuzzing. We find that users lack clear stopping criteria and often rely on intuition or heuristic time limits. While fuzzing is a random process with no "correct" stopping point, building more trust in stopping decisions could help improve efficiency and confidence, especially when users have limited resources.

Finally, as fuzzing expands beyond traditional security domains, future work should investigate usability challenges in details regarding new contexts such as continuous integration, large-scale system testing, or hardware fuzzing. Understanding how workflows, goals, and user expertise differ across domains will be critical to supporting further adoption of fuzzers.

6 Conclusion

Fuzzing continues to evolve as a powerful testing techniques, but its usability remains a critical barriers 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 in 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 by employing better automation, flexibility, actionable feedback, and integration, fuzzing as a testing techniques can be more accessible and effective across a wider range of domains.

References

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

[2] [n. d.]. *Google FuzzTest.* https://github.com/google/fuzztest

[3] Google 2025. *Google/Fuzzbench.* Google. https://github.com/google/fuzzbench

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

[5] 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 *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* 975–985.

[6] Antonia Bertolino. 2007. Software Testing Research: Achievements, Challenges, Dreams. In *Future of Software Engineering (FOSE '07).* 85–103. doi:10.1109/FOSE.2007.25

[7] Paul E Black, Barbara Guttman, and Vadim Okun. 2021. Guidelines on minimum standards for developer verification of software. *arXiv preprint arXiv:2107.12850* (2021).

[8] Marcel Boehme, Cristian Cadar, and Abhik ROYCHOUDHURY. 2021. Fuzzing: Challenges and Reflections. 38, 3 (2021), 79–86. doi:10.1109/MS.2020.3016773

[9] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. 2020. Fuzzing: Challenges and reflections. *IEEE Software* 38, 3 (2020), 79–86.

[10] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-adaptive mutational fuzzing. In *2015 IEEE Symposium on Security and Privacy.* IEEE, 725–741.

[11] Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering.* 332–343.

[12] Victoria Clarke and Virginia Braun. 2017. Thematic analysis. *The journal of positive psychology* 12, 3 (2017), 297–298.

[13] 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.

[14] 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

[15] 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.

[16] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20).* USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi

[17] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN conference on programming language design and implementation.* 206–215.

[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. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 3 (2020), 1–29.

[20] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L Hosking. 2021. Seed selection for successful fuzzing. In *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis.* 230–243.

[21] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. 2020. {FuzzGen}: Automatic fuzzer generation. In *29th USENIX Security Symposium (USENIX Security 20).* 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 *2023 IEEE Symposium on Security and Privacy (SP).* IEEE, 2676–2692.

[23] Ling Jiang, Hengchen Yuan, Mingyuan Wu, Lingming Zhang, and Yuqun Zhang. 2023. Evaluating and improving hybrid fuzzing. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE).* IEEE, 410–422.

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

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

[26] 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 Accessed: 2025-04-13.

[27] 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.*

[28] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18).* Association for Computing Machinery, New York, NY, USA, 2123–2138. doi:10.1145/3243734.3243804

[29] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering.* 475–485.

[30] Jinfeng Li. 2020. Vulnerabilities mapping based on OWASP-SANS: a survey for static application security testing (SAST). *arXiv preprint arXiv:2004.03216* (2020).

[31] 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.

[32] Bingchang Liu, Liang Shi, Zhuhua Cai, and Min Li. 2012. Software Vulnerability Discovery Techniques: A Survey. In *2012 Fourth International Conference on Multimedia Information Networking and Security* (2012-11). 152–156. doi:10.1109/MINES.2012.202

[33] Yuwei Liu, Yanhao Wang, Purui Su, Yuanping Yu, and Xiangkun Jia. 2021. Instruguard: find and fix instrumentation errors for coverage-based greybox fuzzing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 568–580.

[34] Chenyang Lyu, Shouling Ji, Yuwei Li, Junfeng Zhou, Jianhai Chen, and Jing Chen. 2018. Smartseed: Smart seed generation for efficient fuzzing. *arXiv preprint arXiv:1807.02606* (2018).

[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 *28th USENIX security symposium (USENIX security 19).* 1949–1966.

[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.*

[37] Yunlong Lyu, Yuxuan Xie, Peng Chen, and Hao Chen. 2024. Prompt Fuzzing for Fuzz Driver Generation. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security.* 3793–3807.

[38] Reza M. Parizi, Kai Qian, Hossain Shahriar, Fan Wu, and Lixin Tao. 2018. Benchmark Requirements for Assessing Software Security Vulnerability Testing Tools. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, 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 *Proceedings of the 38th Annual Computer Security Applications Conference* (Austin, TX, USA) *(ACSAC '22).* Association for Computing Machinery, New York, 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. *Proceedings of the ACM on human-computer interaction* 3, CSCW (2019), 1–23.

[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 *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis.* 532–543.

[44] Timothy Nosco, Jared Ziegler, Zechariah Clark, Davy Marrero, Todd Finkler, Andrew Barbarello, and W. Michael Petullo. 2020. The Industrial Age of Hacking. In *29th USENIX Security Symposium (USENIX Security 20).* USENIX Association, 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] Saahil Ognawala, Thomas Hutzelmann, Eirini Psallida, and Alexander Pretschner. 2018. Improving function coverage with munch: a hybrid fuzzing and directed symbolic execution approach. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. 1475–1482.

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

[48] Stephan Plöger, Mischa Meier, and Matthew Smith. 2021. A Qualitative Usability Evaluation of the Clang Static Analyzer and libFuzzer with CS Students and CTF Players. In *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*. USENIX Association, 553–572. https://www.usenix.org/conference/soups2021/presentation/ploger

[49] Stephan Plöger, Mischa Meier, and Matthew Smith. 2023. A Usability Evaluation of AFL and libFuzzer with CS Students. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 186, 18 pages. doi:10.1145/3544548.3581178

[50] Roshan Namal Rajapakse, Mansooreh Zahedi, and Muhammad Ali Babar. 2021. An empirical analysis of practitioners' perspectives on security tool integration into devops. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–12.

[51] 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 *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE, 145–154.

[52] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing seed selection for fuzzing. In *23rd USENIX Security Symposium (USENIX Security 14)*. 861–875.

[53] Irving Seidman. 2006. *Interviewing as qualitative research: A guide for researchers in education and the social sciences*. Teachers college press.

[54] Dongdong She, Abhishek Shah, and Suman Jana. 2022. Effective seed scheduling for fuzzing with graph centrality analysis. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2194–2211.

[55] Gabriel Sherman and Stefan Nagy. 2025. No Harness, No Problem: Oracle-guided Harnessing for Auto-generating C API Fuzzing Harnesses. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 775–775.

[56] 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 *Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020)*. 221–238.

[57] 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 *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 248–259.

[58] Murugiah Souppaya, Karen Scarfone, and Donna Dodson. 2022. Secure software development framework (ssdf) version 1.1. *NIST Special Publication* 800, 218 (2022), 800–218.

[59] 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 *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–17.

[60] 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 *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*.

[61] Omer Tripp, Salvatore Guarnieri, Marco Pistoia, and Aleksandr Aravkin. 2014. Aletheia: Improving the usability of static security analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 762–774.

[62] 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.

[63] 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 *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 38–49. doi:10.1109/SANER.2018.8330195

[64] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S. Foster, and Michelle L. Mazurek. 2020. An Observational Investigation of Reverse Engineers' Processes. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1875–1892. https://www.usenix.org/conference/usenixsecurity20/presentation/votipka-observational

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

[66] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 579–594.

[67] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 724–735.

[68] 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*.

[69] 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 *2016 IEEE Symposium on Security and Privacy (SP)*. 158–177. doi:10.1109/SP.2016.18

[70] Qian Yan, Minhuan Huang, and Huayang Cao. 2022. A survey of human-machine collaboration in fuzzing. In *2022 7th IEEE International Conference on Data Science in Cyberspace (DSC)*. IEEE, 375–382.

[71] 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

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

[73] 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 e-prints* (2023), arXiv–2307.

[74] 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 *30th USENIX Security Symposium (USENIX Security 21)*. 2811–2828.

[75] 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 *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1223–1235.

[76] Mingrui Zhang, Jianzhong Liu, Fuchen Ma, Huafeng Zhang, and Yu Jiang. 2021. Intelligen: Automatic driver synthesis for fuzz testing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 318–327.

[77] 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.

[78] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)* 54, 11s (2022), 1–36.

A Appendix: Interview Protocol

Note: The following questions and messages represent the general structure and content of the interview, but may not be asked verbatim. The phrasing or order may be adjusted to fit the flow of the conversation or specific context.

A.1 Introduction

Hi, my name is _______. Thank you for taking the time to participate in our interview study. Our research focuses on understanding how developers and computer security professionals like you use fuzzing tools in your work. We aim to identify both the strengths and challenges of these tools to help guide improvements and make them more effective for the broader computer science community.

Your responses are valuable to us, and we want to assure you that your data will be stored securely. We will not release any sensitive information about you. Importantly, we do not expect you to share any corporate secrets or proprietary information. If at any point you're concerned about disclosing such information, please feel free to skip any questions.

This interview will be transcribed to support later analysis, and all transcripts will be destroyed once the study is complete. We will also share the results of our study with you.

Before we begin, do you have any questions about the study or the consent form you filled out?

A.2 Broad Questions / Background

Before we dive into the main questions, I'll start with a few general questions about your experience using fuzzing tools. These initial questions help us understand the context of your work.

- Can you describe a typical scenario where you've used a fuzzing tool?
  - What has your experience been like in those situations?
  - How frequently do you use fuzzing tools, and in what contexts?
- How did you learn to use fuzzing tools?
  - Were there any resources or people that helped along the way?

A.3 Specific Uses and Challenges

Now that we've covered some background, I'd like to focus on the specific ways you use fuzzing tools and the challenges you may have encountered. This will help us understand the practical aspects of fuzz testing in your work.

- Can you walk me through the steps you follow when setting up and using a fuzzing tool?
  - what's involved in setting up the driver, providing seeds, and analyzing the outputs?
  - Of these tasks, which ones are most time-consuming or challenging, and why?
- What do you think are some good use cases for the fuzzer you use?
  - Have you had to modify or adapt a fuzzer? Can you give an example?
  - Can you share an example of a critical bug found using fuzzing?
  - **If no example**: What types of findings do you consider most valuable?
- What expectations do you have when using a fuzzer? Has it ever failed to meet them?
  - Did you work around the limitation? Use other tools?
- Is there anything you changed about your fuzzing process over time to make it more successful?
- Do you fuzz complex functions? If so, how do you choose them? If not, why?
- What challenges have you experienced integrating fuzzing into other testing processes?
  - Are those challenges related to the tool, workflow, or something else?
  - Do you foresee any learning curve or adoption challenges in getting your team or teammate (intern) to use fuzzing tools effectively? How might you address these challenges?
  - **If no integration**: Do you think integrating fuzzing tools with your existing testing processes (like unit testing, static analysis, or CI/CD) would require significant adjustments? If so, what kind of adjustments would be necessary? What are the challenges?
  - Have these challenges impacted testing efficiency?
  - What part of the tool presents challenges for integration?

A.4 Managing and Interpreting Fuzzer Output

Next, let's talk about how you manage and interpret the results from fuzzing tools. This section is about understanding how you handle the data generated by fuzzers and how you make decisions based on that information.

- At your work, how do you interpret and manage the results generated by a fuzzer?
  - What tools or methods do you use to organize and prioritize fuzzer findings?
  - if they mention specific tools: How effective do you find these tools in helping you manage large volumes of data?
- Do the fuzzing tool you use in your workflow produce a large amount of output?
  - **if yes**:
    * What are some of the biggest challenges you've encountered when managing the large volume of findings generated by fuzzers? How do you usually handle those?
    * Can you give us an example of the challenges you mentioned?
    * Which issues do you address first?
    * What factors influence your prioritization process?
  - **if no**:
    * How do you review the output generated by your fuzzing tool? What are you looking for in these outputs?
    * Do you feel the fuzzing tool provides enough actionable insights even with a limited number of results?
- Have you ever encountered cases where a critical issue was hidden in less important findings? How did you identify it?
  - What strategies do you use to manage duplicate reports?
  - **if they struggle with duplication**: Have you found any tools or methods that help reduce the time spent on managing duplicates?
- How do you decide if you've done enough fuzzing?
  - Do you feel unsure about whether enough fuzz testing has been done? What steps do you take when you're uncertain? Can you give me an example?
  - What are some techniques you use to verify that you have done enough testing
  - **if they mention uncertainty in completion**: How do you handle situations where you're unsure if enough testing has been done?

A.5 Opportunities for Improvement

Finally, I'd like to explore your thoughts on how fuzzing tools could be improved. This section is focused on identifying potential enhancements that could make fuzz testing more effective and efficient in your work.

- In your experience, are there any recurring issues or frustrations with fuzzing tools that you wish could be addressed? What specific improvements would make the most difference in your day-to-day work?
  - Sometimes new features come with trade-offs. Do you think adding these improvements might introduce new challenges, such as increasing complexity or generating too much data to process?

  - **if they suggest a specific feature**: Have you seen any tools that already offer something similar, or would this be a completely new capability?
- What feedback do you think fuzzer can provide to boost your confidence that your program is thoroughly tested?
- How well do you think fuzzing fits with the other tools and testing methods you currently use? Are there any testing tools you'd like to see integrated with your fuzzer to the testing process?
  - What specific capabilities would you want from such an integration?
  - Do you think integrating fuzzers with tools like static analysis or UI-based reverse engineering would make it easier to find and fix issues? How would that look in your workflow?
  - Are there any concerns or challenges you think might arise from integrating these tools?
- If you could create the ultimate fuzzing tool for your work, what are the top three features it would need to have? How would those features solve the biggest issues you face today?
  - **if they struggle to describe features**: Is there a particular problem or inefficiency in your current process that you would want this ideal tool to solve?

A.6 Wrapping Up

Thank you so much for your time and insights today. Your information is incredibly valuable to our study, and we truly appreciate your contribution. Before we wrap up,

- Is there anything else you'd like to share about your experience with fuzzing or anything we haven't covered that you think is important?
- Do you have any questions about this research or anything we've discussed?
- Lastly, if you know anyone else who might be interested in participating in this study, we'd greatly appreciate it if you could refer them to us.

We'll be sure to share the results with you once the study is complete. Thanks again, and have a great day!