Accurate and Low-Cost Residual Risk Assessment via Sampled Profiling and Structure-Aware Coverage Amplification

Seongmin Lee
seongminlee@sigsoft.org
University of California, Los Angeles
Los Angeles, CA, USA

Işıl Özgü
isilozgu@g.ucla.edu
University of California, Los Angeles
Los Angeles, CA, USA

Marcel Böhme
marcel.boehme@acm.org
Max Planck Institute for Security and Privacy
Bochum, Germany

Miryung Kim
miryung@cs.ucla.edu
University of California, Los Angeles
Los Angeles, CA, USA

Abstract

Estimating the *residual risk* of undiscovered bugs—and thus when to stop fuzzing—is critical for determining testing adequacy. Traditionally, this assessment relies on coverage instrumentation, which records *complete coverage* of every execution but imposes a 60-70% performance slowdown, reducing fuzzing throughput and delaying bug discovery. This overhead raises a fundamental question: *can partial observation provide high-fidelity confidence at a fraction of the coverage instrumentation cost?*

We propose a risk assessment method in the context of *black-box fuzzing* that leverages *hardware-enabled performance profiling*—replacing coverage instrumentation with instruction pointer (IP) sampling—as the core mechanism for low-overhead observation. To recover the accuracy inevitably lost to the sparse nature of sampling, we employ *must-execute analysis*, a static control-flow analysis that identifies all statements and basic blocks that *must* have executed to reach, or must execute after, an observed sample, effectively reconstructing missing execution paths from partial observations. Despite the partial coverage inherent to profiler-based sampling, our risk estimation method achieves 4.27× higher throughput than complete-coverage instrumentation, enabling fuzzing campaigns to reach stopping thresholds 2× faster while maintaining high accuracy (1.73 average relative error compared to the ground truth). Ultimately, this efficiency translates into superior bug discovery, triggering more bugs with speedups of up to 10× within the same time budget.

CCS Concepts

• **Software and its engineering** → **Software testing and debugging**; *Dynamic analysis*; • **Theory of computation** → *Program analysis.*

Keywords

fuzzing, residual risk estimation, profiler-based sampling, must-execute analysis, stopping criteria, statistical estimation

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3837519

ACM Reference Format:
Seongmin Lee, Işıl Özgü, Marcel Böhme, and Miryung Kim. 2026. Accurate and Low-Cost Residual Risk Assessment via Sampled Profiling and Structure-Aware Coverage Amplification. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3832783.3837519

1 Introduction

As Dijkstra famously noted, *"testing can be used to show the presence of bugs, but never to show their absence."* To address this fundamental limitation, **residual risk analysis** provides a statistical framework to quantify the likely *absence of errors* remaining after a fuzzing campaign. By applying statistical estimation models to coverage data, recent studies [7–9, 22, 25] can determine when the probability of discovering new bugs has dropped below a desired threshold, facilitating a decision of when to stop fuzzing.

Residual risk analysis typically relies on code instrumentation, which yields complete coverage of every test execution, but this comes with significant performance costs. Source-based instrumentation adds 60-70% latency [11], while binary-only methods can add a 5× to 7× slowdown [11, 27], severely limiting fuzzing throughput. This overhead creates a paradox where the tools used to measure fuzzing adequacy actively impede the fuzzing process they are meant to optimize.

In this paper, we explore the use of *profiler-based sampling*—periodic hardware-based instruction pointer sampling, facilated by modern CPU performance counters [19] and OS-level interfaces like Linux `perf` [3]—to achieve accurate, low-cost fuzzing assessment, in the context of *black-box* fuzzing. By repurposing existing hardware features originally designed for performance analysis [16], our approach samples execution behavior while bypassing the slowdowns of traditional coverage instrumentation. Our core insight is that this partial behavior sampling—while naturally incomplete—can be effectively combined with *white-box, structure-aware residual risk assessment method*. By leveraging the program's control flow graph (CFG) structure, we map sparse samples back to the specific basic-blocks that must have been executed to reach, or after, the sampled point. This *must-execute analysis* is a *sound* mechanism for handling the missing information inherent to profiler-based sampling, recovering significant coverage data without additional

instrumentation. Ultimately, this method enables robust estimation of residual risk without the high tax of coverage instrumentation.

Our study concerns residual risk analysis for *black-box* fuzzing when only a profiler is available and the executable-under-test has not been instrumented for coverage.[1] This problem domain precludes the application of coverage-based greybox fuzzers whose coverage signal needs to be deterministic.[2] Following previous work on residual risk analysis [22], we implement our residual risk analysis in the blackbox variant of AFL++ [14] reusing its mutation operators but disabling all coverage-related features (incl. power schedules and corpus growth; cf. Section 3.3).

We evaluate the fundamental trade-off between the overhead of coverage instrumentation and the precision of residual risk estimation. Specifically, we assess our approach against traditional instrumentation-based methods regarding three key dimensions:

- **Risk Assessment Accuracy**: To what extent does profiler-based sampling—aided by must-execute analysis—approximate the ground-truth residual risk? We quantify the *accuracy cost* of moving from coverage instrumentation to periodic sampling.
- **Fuzzing Efficiency**: What is the performance overhead of profiler-based sampling compared to coverage instrumentation? Does the reduced overhead translate into measurable gains in execution throughput and, consequently, an accelerated rate of unique bug discovery?
- **Time-to-Confidence**: Given a target confidence level for testing adequacy, how much earlier (in wall-clock time) can a sampling-based campaign reach the stop threshold compared to a coverage-instrumented fuzzing campaign?

We conduct a systematic experimental study across a diverse set of real-world programs to evaluate these trade-offs. Across FuzzBench [26] and Magma [17] benchmarks, our results demonstrate a compelling trade-off: must-execute analysis reduces the overestimation of residual risk from nearly an order of magnitude (8.43 relative error) to within the same order of magnitude as the ground truth (1.73 relative error), providing high accuracy despite partial observation. In terms of efficiency, the minimal overhead of profiler-based sampling translates to 4.27× higher throughput, discovering 19 bugs within the same time limit, as opposed to 16 bugs with coverage instrumentation, while finding them up to 10× faster. Consequently, our approach yields significant time savings to reach the given confidence level, reaching stopping criteria earlier in 12 out of 19 subject programs—up to 2× faster than coverage instrumentation.

The key contributions of this paper are as follows:

- While *coverage instrumentation* is the standard approach for fuzzing despite its overhead, we are the first to design a residual risk assessment method using hardware IP sampling and white-box CFG analysis. We demonstrate that developers can maintain high confidence in fuzzing adequacy by sampling coverage non-invasively at intervals, providing an accurate risk assessment even with incomplete data.
- By incorporating static inter-procedural analysis, we reduce the relative error from 8.43 to 1.73, showing that white-box, structure-aware coverage inference is essential for maintaining confidence in fuzzing adequacy. This improvement demonstrates that our method provides a sound, principled means to account for missing observations by leveraging the program's inherent control-flow dependencies.
- By removing the overhead of coverage instrumentation, our method significantly improves fuzzing throughput, which directly accelerates vulnerability discovery and enables the early termination of fuzzing campaigns. Removing coverage instrumentation yields 1.3×–7.1× higher throughput. For instance, `sqlite3` (7.1× throughput gain) achieves up to a 10× faster mean time-to-trigger (TTT).

Our findings suggest that coverage instrumentation can be effectively obviated for assessing fuzzing adequacy. By leveraging existing hardware and OS-level sampling mechanisms alongside our principled, white-box program analysis, we provide a sound means to account for missing program behavior observations without the cost of coverage instrumentation. Ultimately, profiler-based sampling empowers developers to either reach a desired confidence threshold significantly faster or, conversely, discover a greater number of bugs within the same fixed time budget.

2 Motivation and Background

Motivation. Testing cannot prove the absence of bugs, leaving practitioners without a principled stopping criterion for fuzzing. Residual risk analysis allows to quantify the likelihood that an undiscovered bug can still be found within an ongoing fuzzing campaign by treating test executions as statistical evidence [7]. Hence, a practitioner can stop a fuzzing campaign when further execution yields diminishing returns. Prior work [7, 9, 22] models estimate residual risk through black-box discovery probability (DP). Each execution is a sample of coverage elements and exposed bugs; given n runs, DP represents the likelihood that the next execution reveals an unseen coverage element. This DP serves as an upper bound for residual risk.

Residual risk analysis is often used in practice. For instance, the fuzzing dashboard developed by Consensys Diligence features a residual risk estimate as a measure of progress of their smart contract fuzzer.[3] The project lead of Microsoft's Project OneFuzz, a continuous fuzzing platform, is planning to use residual risk estimates to "intelligently terminate jobs as the likelihood of finding additional bugs drops" rather than run for a fixed duration.[4]

A residual risk estimate yields a principled stopping rule for fuzzing: halt once the estimated discovery probability drops below a threshold ε. The practitioner sets ε directly as the probability they will tolerate that the next input still reveals unseen behavior (say, one in a million). Unlike a fixed time budget, this criterion tracks a campaign's actual progress rather than the clock: a still-productive fuzzer keeps running while a saturated one stops, so two campaigns reach the same ε at different times. A time budget therefore *follows* from ε and the achievable throughput (via the $1/\varepsilon$

[1] While there exist fleet-wide coverage profilers that can profile *any* running program [20], enabling coverage-based fuzzing would require every third-party program to be decompiled and reinstrumented, which is impractical, e.g., for supply chain analysis.
[2] We discuss an extension to greybox fuzzing in Section 7.

[3] https://fuzzing-docs.diligence.tools/general/reports/coverage
[4] https://x.com/metr0/status/1396107664669634563

executions expected until the next discovery) rather than serving as a proxy for it.

Lee and Böhme [22] proposed a dependency-aware estimator that provides a tighter upper bound on discovery probability by grouping coverage elements that co-occur in the same execution. Unlike the standard Good-Turing estimator [15], their method accounts for internal dependencies to yield a more accurate risk estimate. Prior risk estimation methods [7, 9, 22] rely on having full-coverage information for every test execution, creating a significant performance bottleneck.

Coverage as Fuzzing Feedback. Code coverage serves as the de-facto mechanism for fuzzing feedback and is collected via source-level instrumentation or binary rewriting. Both incur significant latency overhead: source-instrumented binaries suffer 60–70% slowdown compared to un-instrumented execution [11], and dynamic binary instrumentation [10] incurs even higher costs, up to 5–7× slowdown [27]. This overhead of running instrumented code reduces fuzzing throughput—fewer test cases can be executed per unit time—undermining the original purpose of bug discovery and fuzzing adequacy assessment. Several lines of work attempt to mitigate this overhead. Lightweight instrumentation strategies such as INSTRIM [18] reduce the number of instrumentation points by exploiting control-flow dominance relations, lowering overhead with the tradeoff of coarser coverage granularity. Full-speed fuzzing [27] runs an uninstrumented binary by default and only activates tracing when new coverage is detected, amortizing the overhead over many executions. Hardware-assisted approaches such as kAFL [31] and the hardware extensions proposed by Ding et al. [11] leverage *Intel Processor Trace* or custom hardware to collect coverage with lower overhead. Unlike our approach, existing approaches lack a principled mechanism for handling incomplete observations inherent to periodic sampling, while accounting for dependencies between observed and unobserved coverage, entailed by control-flow.

Sampling Profilers. A sampling profiler (a profiler or profiling for short) periodically captures snapshots of a program's state. At set intervals, it records the current instruction pointer (IP) and, often, the call stack, creating a statistical timeline of execution. On Linux, perf [3] offers a unified interface for IP sampling on x86, ARM, and RISC-V [12]. Similar capabilities exist via DTrace (macOS/Solaris), Windows Performance Analyzer, and browser-based tools like the Firefox Profiler, making sampling profilers a universally applicable observation mechanism.

Sampling-based profilers achieve near-zero overhead—typically less than 2% for standard frequencies [6]—because they sample the IP without instrumentation. This cost is largely independent of program complexity and remains orders of magnitude lower than coverage instrumentation. Furthermore, the overhead is highly tunable: lowering the sampling frequency further reduces performance costs at the expense of sample density.

3 Residual Risk with Profiler-based Sampling

Our goal is to achieve high-confidence fuzzing adequacy assessment, while avoiding coverage instrumentation overhead. Figure 1 illustrates the workflow of our approach. Instead of employing coverage instrumentation—which yields full coverage data, we utilize

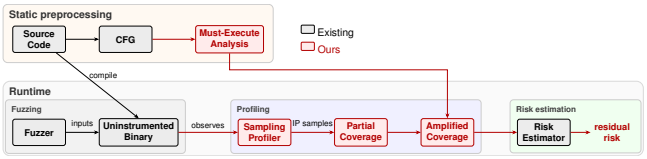


Figure 1: Overview of our approach.

Table 1: Sampled stack traces (ST1–ST4) vs. full coverage (FC1, FC2) over two possible test executions. ■ covered, ■ sampled IP, ■ caller frames.

L#	Code	Execution with flag=True FC1	ST1	ST2	ST3	Execution with flag=False FC2	ST4
1	`void f4() {`					–	
2	`  instruction1;`		Top			–	
3	`  return;`					–	
4	`}`						
5	`void f3() {`					–	
6	`  instruction1;`					–	
7	`  instruction2;`			Top		–	
8	`  return;`					–	
9	`}`						
10	`void f2() {`						
11	`  instruction3;`				Top		Top
12	`  return;`						
13	`}`						
14	`void f1(flag) {`						
15	`  f2();`						
16	`  if (flag) {`					–	
17	`    f3();`					–	
18	`    f4();`					–	
19	`  }`						
20	`  return;`						
21	`}`						
22	`void main(input) {`						
23	`  flag = input % 2;`						
24	`  ret = f1(flag);`						
25	`}`						

a sampling profiler to collect instruction pointer samples at regular intervals. The sampling frequency acts as a tunable knob that controls the trade-off between observation granularity and overhead: higher frequencies yield detailed coverage information at the cost of increased profiling overhead, while lower frequencies reduce overhead but provide coarser coverage data. We analyze the impact of incomplete nature of partial-coverage data on the existing risk estimator [22] and identify the bias it introduces (§3.1). To mitigate this bias, we design a sound *must-execute analysis* (§3.2) that operates in tandem with the estimator to boost coverage data based on intra- and inter- procedural dependencies entailed by a control flow graph. While this paper concerns the problem of assessing fuzzing adequacy, the mechanism of amplifying incomplete coverage information using a must-execute analysis could be also used in the context of feedback-guided fuzzing, by replacing full-coverage signal with a feedback signal from a profiling sampler. The implication of leveraging a sampling profiler is detailed in Section 7.

3.1 Estimation with Partial Coverage

Sampling profilers yield an incomplete set of instruction pointers (IPs) that capture only a fraction of program behavior during a fuzzing campaign. As shown in Table 1, the resulting profile (stack traces ST1–4 over two test executions) is sparser than the complete coverage recorded by instrumentation (FC1, FC2), especially when sampling only the top-of-stack IP rather than the full call stack.

This partial-coverage data introduces bias in existing singleton-based estimators of residual risk—the probability that undiscovered

bugs remain in the program. Existing estimators[5] count test executions that contain a *unique coverage element* (basic block, line, IP, etc.) observed in exactly one execution—termed a *singleton*. Specifically, the dependency-aware estimator by Lee and Böhme [22]:

$$\hat{GT} = \frac{V_1^{\equiv}}{n}, \tag{1}$$

where n is the number of test executions and $V_1^{\equiv}$ is the number of such singleton-containing executions. Under full-coverage information, this produces statistically consistent estimates [22].[6] Under profiler-based sampling, however, the estimator receives incomplete coverage per execution, which distorts $V_1^{\equiv}$ in two ways:

- *Case 1 (over-estimation): Different partial coverage from executions with the same coverage.* Due to the non-deterministic nature of a sampling profiler, executions with identical coverage can produce different profiler samples. In Table 1, consider two executions both with full coverage FC1, where the profiler produces ST1 from one and ST2 from the other. Under full coverage, both executions record identical elements—no singletons. Under partial coverage, ST1 and ST2 capture different elements (e.g., line 2 vs. line 7); elements unique to one sample become false singletons that inflate $V_1^{\equiv}$ and overestimate residual risk.
- *Case 2 (under-estimation): Same partial coverage from executions with different coverage.* Conversely, executions with different coverage can produce identical profiler samples. In Table 1, consider an execution with full coverage FC1 and another with FC2, where the profiler produces ST3 from the first and ST4 from the second. Under full coverage, the FC1 execution contains elements not in FC2 that are correctly counted as singletons. Under partial coverage, ST3 and ST4 capture identical elements, so neither execution contributes a singleton, deflating $V_1^{\equiv}$ and underestimating residual risk.

Under-estimation is more problematic than over-estimation in a security context, as it fosters a false sense of confidence, causing to stop fuzzing prematurely, when vulnerabilities remain undiscovered. Conversely, over-estimation—though less dangerous—results in inefficient resource allocation by prolonging a fuzzing campaign.

We hypothesize that over-estimation (Case 1) is more likely to occur than under-estimation (Case 2) for moderate sampling frequencies. Under partial coverage, a significantly large number of distinct partial-coverage observations can arise from the same execution, while the same partial coverage appearing across different executions is less likely. We formalize this intuition below using mathematical modeling and also empirically test this hypothesis in our experiments (§4).

Mathematical Model of Estimation Error. This section formalizes how profiler-based sampling affects the estimator in Eqn. (1).

The estimator operates on the coverage recorded for each execution. Under coverage instrumentation, every exercised element is recorded; under profiler-based sampling, only elements at sampled instruction pointers are recorded. Consider S coverage elements, each appearing independently across m executions. For any arbitrary execution, the probability that element i appears in an execution's recorded coverage is p_i under coverage instrumentation and $q_i \le p_i$ under profiler-based sampling.

The expected probability g that element i is a singleton (observed in exactly one execution) in m executions is, for observation probability $p \in \{p_i, q_i\}$,

$$g(p, m) = m\,p\,(1-p)^{m-1},$$

and the expected discovery probability under full and partial coverage follows from summing g over all S elements and dividing by m:

$$\hat{GT}_{\text{full}} = \frac{1}{m}\sum_{i=1}^{S} g(p_i, m), \qquad \hat{GT}_{\text{partial}} = \frac{1}{m}\sum_{i=1}^{S} g(q_i, m).$$

The difference between $\hat{GT}_{\text{full}}$ and $\hat{GT}_{\text{partial}}$ depends on how g behaves differently for p_i vs. q_i:

Proposition 3.1. *The singleton probability $g(p, m) = m\,p\,(1-p)^{m-1}$ has the following properties:*

(1) ***Peak and decay.*** *g rises to a single peak at $m^*(p) = -1/\ln(1-p) \approx 1/p$ for small p, then decays to zero for larger m.*
(2) ***Peak shift.*** *For $q \le p$, $m^*(q) \ge m^*(p)$.*
(3) ***Similar peak height.*** *$g(p, m^*(p)) \to 1/e$ as $p \to 0$.*
(4) ***Slower decay.*** *For $q \le p$, $g(q, m)$ decays more slowly than $g(p, m)$ since $(1-q) > (1-p)$.*

Since $q_i \le p_i$, properties (2)–(4) together imply that $g(q_i, m)$ remains elevated over a wider range of m than $g(p_i, m)$. This leads to two key predictions.

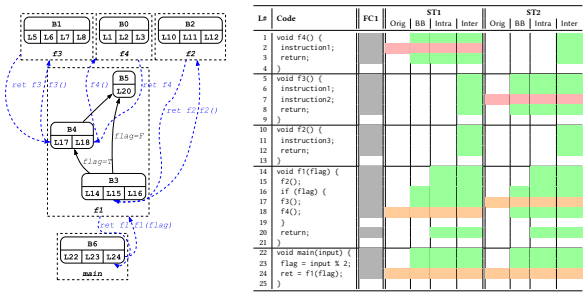
First, partial coverage tends to over-estimate the discovery probability ($\hat{GT}_{\text{partial}} > \hat{GT}_{\text{full}}$). This shows that our risk estimation likely be conservative and avoids providing false confidence (i.e., our tool might tell you to keep testing longer than necessary, but it is less likely to tell you to stop too early.) Under partial coverage, each element remains a plausible singleton over a wider range of execution counts than under full coverage; thus, $\hat{GT}_{\text{partial}}$ tends to count more singletons than $\hat{GT}_{\text{full}}$, producing a higher (more cautious) estimate.

Second, the estimation error decreases as coverage saturates. This shows that our risk estimation accuracy improves as more data is collected. As a campaign progresses, both $g(p_i, m)$ and $g(q_i, m)$ move past their peaks and decay toward zero, so the gap between $\hat{GT}_{\text{full}}$ and $\hat{GT}_{\text{partial}}$ narrows over time.

3.2 Must-Execute Analysis

To mitigate the estimation error caused by partial coverage information, we apply *must-execute analysis*: given the observed samples, we use classical dominator and post-dominator analysis, extended across procedure boundaries, to identify instructions that must also have executed. By analyzing the control-flow graph of the program, we can infer additional coverage information beyond what is directly observed from the profiler samples. This allows us to refine our estimate of residual risk by accounting for unobserved instructions that must have been executed based on the observed samples.

[5] Adopted from ecology, where they estimate the probability of discovering a new species.
[6] That is, the estimate converges to the true discovery probability as the number of executions increases.



(a) Control-flow graph **(b) Coverage expansion**

Figure 2: Must-execute analysis: (a) control-flow graph and (b) coverage amplification at three levels

We divide must-execute analysis into three levels of granularity: basic block level, intra-procedural control-flow level, and inter-procedural control-flow level.

(1) **Basic block level (BB)**: We infer that all instructions within the same basic block must have executed, if at least one instruction in that block was observed.
(2) **Intra-procedural control-flow level (Intra)**: We analyze the control-flow graph of a single procedure to infer additional coverage from dominance relations. If an instruction n is observed, all *dominators* of n must have also executed, since every path from the procedure entry to n passes through them. Similarly, all *post-dominators* of n must also execute, provided the program terminates normally or reaches the procedure exit.
(3) **Inter-procedural control-flow level (Inter)**: We extend the dominance analysis across procedure boundaries to infer that if any call site is either observed or inferred to have executed, then we infer the execution of the callee's entry instruction; vice versa, if any instruction within a procedure is observed, its entry point is necessarily executed.

The analysis is a straightforward application of standard dominator and post-dominator analysis, which amplifies sparse profiler-IP samples to reduce the estimation error caused by partial observation. Because we estimate residual risk from the inferred coverage, the analysis must be *sound*: we infer that an instruction executed only when it provably did. Unsound over-inclusion would deflate the singleton count and *underestimate* residual risk, which is dangerous because it can stop a campaign prematurely. For indirect calls through function pointers, we therefore take a sound under-approximation that leaves unresolved call edges out, keeping the estimate conservative. Because the analysis runs once, before fuzzing, a more precise resolver could be substituted at no fuzzing-time cost.

Figure 2 illustrates an example of must-execute analysis applied to two stack trace samples, ST1 and ST2, extracted from the same execution with full coverage FC1, of a simple program from Table 1. Given the control-flow graph in Figure 2a, we apply must-execute analysis at different levels of granularity. The coverage expansion table in Figure 2b shows how the original partial coverage (ST1, ST2) is expanded to include additional instructions that are inferred to have executed based on the control-flow analysis. In this example, after applying inter-procedural control-flow analysis, the expanded coverage from ST1 and ST2 become identical to the full coverage FC1. This demonstrates the potential of must-execute analysis to mitigate estimation errors inherent to partial coverage data.

3.3 Implementation

Black-box fuzzing configuration. We implemented our approach on top of the residual risk analysis framework of Lee and Böhme [22], using their risk estimator to compare the accuracy and efficiency of profiler-based sampling against coverage instrumentation. Their framework is a black-box fuzzer built on AFL++: it disables coverage-guided feedback loop. Specifically, it disables (a) the queuing of new-coverage inputs (`IGNORE_FINDS`, `save_if_interesting`), (b) favored/redundant seed scheduling and culling, (c) the repeated stacking of `havoc` mutations, and (d) time-dependent changes to the mutation strategy. As a result, each input is generated independently of the coverage observed so far, which is the defining behavior of a black-box fuzzer. Dedicated black-box fuzzers (e.g., Radamsa [1], zzuf [4], Peach [2]) predate coverage-guided fuzzing and are not integrated with modern benchmarks such as FuzzBench [26] and Magma [17], making the black-box configuration of AFL++ the natural choice for our evaluation. Since a black-box fuzzer is characterized solely by the independence of successive executions from observed coverage, the choice of underlying fuzzing engine does not affect the black-box behavior itself.

Profiler-based sampling. We use Linux `perf` to collect instruction pointer (IP) samples at regular intervals during fuzzing campaigns. Each sample triggers a system interrupt that incurs overhead, so the sampling frequency controls the trade-off between observation granularity and performance: higher frequencies yield more detailed information at the cost of increased overhead, while lower frequencies reduce overhead but provide coarser data. To investigate this trade-off, we set the sampling frequency relative to the executions per second of the fuzzing campaign to maintain a consistent observation (e.g., X samples per test execution on average) ratio across different subjects and configurations, (default f=1: one sample per execution).

By default, `perf` captures a single IP at each sampling point; we call this top mode. `perf` also supports capturing the full call stack via the `-call-graph` option, which records the chain of return addresses from the current instruction pointer to the top-level caller; we call this stack mode. stack mode yields multiple observed IPs per sample, providing richer coverage data, but requires the binary to be compiled with frame pointers preserved (`-fno-omit-frame-pointer`) and incurs higher runtime overhead due to the cost of unwinding and processing full call stacks. We evaluate both configurations to understand the trade-offs between overhead and accuracy. Since AFL++ normally forks a new child process for each execution, we use its persistent mode to keep the target process alive across multiple iterations, allowing `perf` to remain attached to a stable process. We employ `perf record` to collect samples and `perf script` to extract IP data, pipelining these two steps to avoid I/O overhead.

Per-execution discovery probability. Because profiler-based sampling collects observations at fixed time intervals rather than per execution, the resulting estimation yields a per-second discovery probability $\hat{p}$. To obtain the per-execution discovery probability,

Table 2: Subject program statistics.

Subject	Description	Commit	Harness	NCLOC	#Basic Blocks
proj4	Cartographic projections	a7482d3	CRS-to-CRS coordinate transformation	407,430	31,747
sqlite3	SQL database engine	c78cbf2	SQL parsing and execution	255,512	38,161
mbedtls	Crypto library for embedded	169d9e6	SSL/TLS handshake simulation	211,562	4,596
libxml2	XML parser	c7260a4	XML parsing from memory	211,267	48,036
freetype2	Font rendering library	cd02d35	Font parsing, glyph loading	118,830	24,037
harfbuzz	Text shaping engine	cb47dca	Glyph drawing, OpenType features	117,789	15,508
libpcap	Packet capture library	17ff63e	BPF filter compilation, packet parsing	47,824	5,961

which is the standard metric for residual risk estimation [22, 25], we convert as $\hat{p}_{\text{exec}} = 1 - (1 - \hat{p})^{1/n}$, where n is the average throughput (exec/s). This conversion assumes independent discoveries across executions, an assumption that holds because coverage-guided feedback is not used to influence subsequent test executions. We also empirically verified that throughput remains stable within each sampling interval.

Must-execute analysis. We implement must-execute analysis as a static preprocessing step. We compile the target program with Clang to generate LLVM IR and construct the control-flow graph. We then extract inter-procedural must-execute relationships by analyzing function call edges and store them in a database for efficient querying during fuzzing campaigns. At runtime, we use `addr2line` to map the IP samples collected by `perf` to source-code locations and query the must-execute database to expand the observed partial coverage.

4 Evaluation

We evaluate the impact of replacing coverage instrumentation with profiler-based sampling for residual risk estimation, focusing on three key dimensions:

- *Estimation accuracy.* Profiler-based sampling provides less complete coverage data than coverage instrumentation. As discussed in Section 3.1, this incompleteness tends to result in a conservative over-estimation of residual risk.
- *Bug discovery.* Profiler-based sampling significantly reduces coverage-tracking overhead, increasing fuzzing throughput. We expect this higher execution rate to accelerate vulnerability discovery, finding more bugs within a given time budget.
- *Time-to-confidence.* These two effects pull in opposite directions. While overestimation of residual risk tends to prolong a fuzzing campaign, higher throughput reduces the ground-truth residual risk faster, potentially enabling an earlier stopping decision overall. We evaluate how these competing effects balance out.

We systematically evaluate these trade-offs across various sampling configurations (frequency, mode) and must-execute analysis levels. To account for the stochastic nature of fuzzing, we performed five repetitions of each experiment.

Subject programs. We select seven real-world subjects from the FuzzBench framework [26], prioritizing large-scale programs where instrumentation overhead is most significant. Four of these subjects are drawn from prior work [22], and three are additionally included from FuzzBench. These subjects span various domains—cryptography, parsing, rendering, and networking—to ensure generalizability. Table 2 summarizes the subject characteristics, including descriptions, commit versions, and code metrics (NCLOC and basic blocks).

For §4.2's the bug analysis (**RQ2**), we need a benchmark with known vulnerabilities and exploits. We use Magma [17], a fuzzing benchmark that forward-ports real bugs from historical CVEs and bug reports into the latest version of each target program. By injecting lightweight instrumentation at the bug sites, Magma can detect when a fuzzer *reaches* a bug (execute the vulnerable code)—and when it *triggers* the bug (evidences the vulnerability's symptom such as a crash). Among the nine subjects in Magma, we select libxml2 and sqlite3, which have 2 and 1 fuzz drivers respectively to maintain overlap with the first part of our evaluation. To broaden our scope, we also include openssl that features the highest number of fuzz drivers (6) in Magma. Table 3 summarizes the fuzz drivers and the number of forward-ported bugs for each subject.

Table 3: Magma subjects and fuzz drivers for bug analysis.

Subject	Fuzz Drivers	#Bugs
openssl	`asn1`, `asn1parse`, `bignum`, `server`, `client`, `x509`	20
libxml2	`libxml2_xml_read_memory_fuzzer`, `xmllint`	17
sqlite3	`sqlite3_fuzzy`	20

Environment. All experiments run on an AMD EPYC 9655P 96-Core Processor at 2.6 GHz with 377 GB of RAM, running Ubuntu 22.04 LTS inside Docker. We compile all subjects with Clang/LLVM 14. We use AFL++ v4.33c as the fuzzing engine, Linux `perf` 6.8 for profiler-based sampling, and `addr2line` from GNU Binutils 2.38 for address-to-source mapping.

4.1 Accuracy of Residual Risk Estimation

RQ1: *How accurate is the residual risk estimation using profiler-based sampling?* We compare the accuracy of profiler-based residual risk estimation against the empirical ground truth from coverage instrumentation, while varying the level of must-execute analysis.

To obtain accurate measurements, we compute the estimates and the ground truth from the same fuzzing campaign, so that both are derived from the same sequence of inputs. We compute the ground truth discovery probability at time T using prior work [22]: it is defined as the number of executions that yield new coverage between time $T+1$ and $2T$, divided by the total executions within that interval. Computing this ground truth requires coverage instrumentation. Therefore, each evaluation campaign is run with both the sampling profiler and coverage instrumentation enabled simultaneously; from this instrumentation data, we also compute the full coverage-based residual risk estimate, as shown in Table 4. This dual-collection setup is for evaluation purposes only and is not intended for practical deployment.

We measure accuracy using the relative absolute error [22]: $RE^{r}_{\{esti\}} = |\hat{GT}_{\{esti\}} - \hat{r}_{emp}|/\hat{r}_{emp}$, where $\hat{GT}_{\{esti\}}$ is the estimated residual risk (using either profiler-based sampling or coverage instrumentation), and $\hat{r}_{emp}$ is the empirical discovery probability. A value of 0 indicates a perfect estimate, while 1 indicates an error of 100% relative to the ground truth. We analyze how the relative absolute error varies across different must-execute analysis levels and time intervals during the fuzzing campaign. We fix the sampling frequency to the default ($f = 1$) and use top mode to isolate the effect of the estimation method and must-execute analysis.

4.1.1 Results. Figure 3 shows the estimated discovery probability over time for each subject, comparing profiler-based sampling at

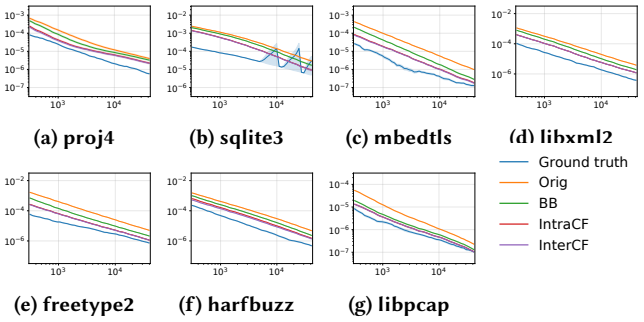


Figure 3: Discovery probability over time (log-log) for each subject, comparing ground truth against risk estimates from a sampling profiler with must-execute analysis. Shaded regions: ±1 standard error.

Table 4: Relative error of estimated residual risk by sampled profiles (Profiling) against empirical (GroundTruth): without must-execute analysis (Orig) and with three must-execute analysis levels (BB, Intra, and Inter). Δ_{Orig} is the absolute estimation difference between Orig and Instrumentation.

Subject	Exec/s	$\hat{r}_{emp}$@12h	Relative Error of Profiling				Relative Error of Instrumentation	Δ_{Orig}
			Orig	BB	Intra	Inter		
libpcap	7,477	1.02e-07	2.14	0.68	0.38	0.37	0.27	1.90e-07
mbedtls	1,767	1.23e-07	15.64	4.52	2.12	2.00	0.61	1.85e-06
libxml2	2,199	3.72e-07	10.05	4.74	2.35	2.33	0.23	3.66e-06
harfbuzz	1,202	4.71e-07	11.04	5.31	2.95	2.48	0.33	5.05e-06
proj4	1,763	5.78e-07	5.57	4.01	2.54	2.39	0.22	3.09e-06
freetype2	1,944	7.11e-07	7.93	2.51	0.87	0.81	0.19	5.50e-06
sqlite3	1,259	3.15e-05	6.65	4.03	1.83	1.76	0.13	2.06e-04
Average			8.43	3.69	1.86	1.73	0.28	

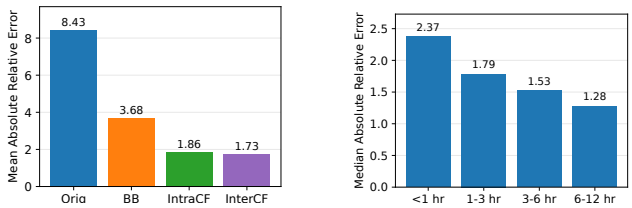


Figure 4: Average relative error. Left: Inter-procedural analysis reduces the average estimation error from 8.43 to 1.73. Right: As the fuzzing duration increases from 1 to 12 hours, the median risk estimation error decreases from 2.37 to 1.28.

different must-execute analysis levels against the ground truth. Each line represents the average across multiple runs, with error bars indicating standard deviation. Both axes use a logarithmic scale. We report results up to 12 hours because each campaign runs for 24 hours; the latter 12 hours are used to compute the empirical ground-truth discovery probability.

Across all subjects, without must-execute analysis and with three different must-execute analysis levels, profiler-based sampling consistently over-estimates the discovery probability relative to the ground truth; this confirms our hypothesis that profiler-based sampling does not lead to overconfidence of prematurely stopping fuzzing, as predicted by our mathematical model in Section 3.1.

Table 4 and Figure 4 (left) show how must-execute analysis progressively reduces estimation error. Without any coverage inference (Orig), each IP sample is attributed to a single instruction, yielding an average *RE* of 8.43. Basic-block propagation (BB) provides the largest single reduction (to 3.69), and further control-flow analysis narrows the error to 1.86 (Intra) and 1.73 (Inter). This is achieved without any coverage instrumentation—the profiler observes only sparse IP samples—yet the estimation error remains well within one order of magnitude of the instrumentation-based estimate (0.28), demonstrating that must-execute analysis can effectively amplify coverage data from sampled profiles alone.

Figure 4 (right) shows the median relative error across different time intervals. The error is highest in the first hour (2.37), when sparse samples leave the coverage estimate most incomplete. However, it decreases monotonically as the campaign progresses: 1.79 (1–3 hours), 1.53 (3–6 hours), and 1.28 (6–12 hours). This consistent improvement indicates that profiler-based sampling becomes increasingly reliable as accumulated samples narrow the gap with instrumentation-based estimation.

Subject-specific observations. The last column of Table 4 shows Δ_{Orig}, the absolute difference between the profiler-based (Orig) and instrumentation-based estimates. As predicted by the mathematical model in Section 3.1, this gap correlates with empirical discovery probability $\hat{r}_{emp}$: it is larger when the fuzzer is actively discovering new coverage and smaller as it approaches saturation. For instance, sqlite3 has both the highest $\hat{r}_{emp}$ and the largest Δ_{Orig}, while libpcap has both the lowest. This confirms the prediction in Section 3.1 that the estimation gap narrows naturally as coverage saturates.

Among must-execute analysis levels, mbedtls is notable: while its *RE* at Orig (15.64) follows the general trend, but basic-block coverage amplification reduces *RE* to 4.52—the largest reduction among all subjects—suggesting that its code structure is particularly well-suited for amplifying coverage through basic-block level, control-flow analysis.

Statistical significance. We apply statistical tests over the observed differences in *RE* across must-execute analysis levels. Because *RE* distributions are non-normal, we apply the non-parametric Mann-Whitney U test with the Vargha–Delaney $\hat{A}_{12}$ as effect size, where $\hat{A}_{12} > 0.71$ indicates a large effect size [32]. Moving from Orig to BB and from BB to Intra must-execute analysis levels reduces *RE* significantly for all seven subjects ($\hat{A}_{12} \geq 0.875; p < 0.001$). At the finest level, Inter shows no significant improvement over Intra for five of seven subjects ($p > 0.05$). Detailed statistical results are in the replication package (Section 8).

> ***Summary (RQ1, accuracy):*** Profiler-based sampling with inter-procedural must execute analysis reduces the relative error from 8.43 to 1.73 and estimation accuracy improves with longer campaign durations. The results confirm that risk estimates avoid overconfidence—consistent with the mathematical model in Section 3.1—demonstrating that sparse sampling provides a rigorous, reliable substitute for coverage instrumentation.

4.2 Efficiency and Vulnerability Discovery

RQ2: *How much does profiler-based sampling speed up fuzzing throughput, compared to coverage instrumentation?* We also assess

Table 5: Bug discovery on Magma subjects over twenty runs of a 24-hour fuzzing campaign. Triggers: number of runs triggering the bug (out of 20). Mean TTT (time-to-trigger) where untriggered runs contribute 24 h). Bold indicates the shorter TTT.

		exec/s		Trig (/20)		Mean TTT	
Subject / Driver	Bug	Instrumentation	Profiling	Instrumentation	Profiling	Instrumentation	Profiling
libxml2 / `xml_read_memory`	XML002	1,414	5,294	0	1	–	**23h39m**
	XML003			20	20	1h16m	**12m34s**
	XML009			20	20	41m43s	**7m55s**
	XML017			20	20	1m	1m
libxml2 / `xmllint`	XML002	1,175	2,467	0	1	–	**22h56m**
	XML009			20	20	23m45s	**20m**
	XML017			20	20	1m	1m
openssl / `asn1`	SSL001	116	191	20	20	3h19m	**1h52m**
	SSL003			20	20	2m22s	**1m30s**
openssl/ `asn1parse`	SSL010	5,096	20,250	0	0	–	–
openssl/ `bignum`	SSL010	27	34	0	0	–	–
openssl / `client`	SSL002	275	626	20	20	1m45s	**1m15s**
openssl / `server`	SSL002	82	187	20	20	3m15s	**1m52s**
	SSL020			11	18	15h51m	**7h58m**
openssl / `x509`	SSL009	2,032	6,904	20	20	4h29m	**1h20m**
sqlite3 / `sqlite3_fuzz`	SQL002	1,911	13,628	20	20	3h25m	**20m**
	SQL003			0	6	–	**20h15m**
	SQL012			7	20	20h35m	**4h07m**
	SQL013			7	19	19h08m	**6h56m**
	SQL014			15	20	15h29m	**2h29m**
	SQL018			20	20	15m36s	**4m**

how the throughput improvements from profiler-based sampling translate into faster bug discovery.

We separately run fuzzing campaigns with coverage instrumentation (Instrumentation mode) and profiler-based sampling (Profiling mode). For throughput, we measure executions per second (exec/s). For vulnerability discovery, we compute the Time-to-Reach (TTR) and Time-to-Trigger (TTT) for each vulnerability under both configurations. To determine when a vulnerability is reached or triggered, we rely on Magma's canary-based monitoring. This provides how many runs reached or triggered each bug across every fixed 30 second interval. We fix the must-execute analysis level to interprocedural, as it provides the highest estimation accuracy, without affecting throughput.

4.2.1 Bug analysis result on Magma. Table 5 shows the bug discovery results for the 19 bug–driver combinations triggered by either coverage instrumentation (Instrumentation) or profiling (Profiling) with frequency $f = 1$ and top mode on three Magma subjects, averaged over twenty independent 24-hour campaigns. For each bug, we report the number of triggered campaigns and the mean time-to-trigger (TTT); runs failing to trigger the bug contribute the full 24-hour maximum. We bold the shorter mean TTT between the two modes. The table also includes the `asn1parse` and `bignum` drivers, which did not trigger any bugs in either mode, to provide a complete throughput comparison across all nine fuzz drivers. We omit Time-to-Reach (TTR) from the table; the complete table listing all 56 bug–driver combinations reached by at least one mode across the nine fuzz drivers is available in the shared artifact (§8). Of these 56 combinations, 50 are reached by both modes in all twenty runs. For libxml2 and openssl subjects the mean TTR is uniformly within one minute. sqlite3 bugs exhibit longer and more variable TTR (e.g., SQL012: Instrumentation 6h35m vs. Profiling 51m48s; SQL013: 8h26m vs. 1h31m), reflecting the throughput advantage of Profiling, but these differences are subsumed by the TTT analysis.

Across all nine fuzz drivers, Profiling achieves 1.3× to 7.1× higher throughput than Instrumentation, with the largest speedup on sqlite3 (1,911 vs. 13,628 exec/s, 7.1×) and libxml2/`xml_read_memory` (1,414 vs. 5,294 exec/s, 3.7×). Even the smallest gain, on openssl / `bignum` (27 vs. 34 exec/s, 1.3×), shows that removing coverage instrumentation consistently improves fuzzing throughput.

Higher throughput under Profiling directly translates into faster and more reliable bug discovery. Across twenty runs, Profiling triggers 19 unique bug–driver instances compared to 16 for Instrumentation, capturing three instances that Instrumentation never finds: `XML002` (via both `xml_read_memory` and `xmllint`) and `SQL003`. Profiling also demonstrates superior consistency. For example, it triggers `SSL020` in 18/20 runs (vs. 11/20 for Instrumentation) and `SQL013` in 19/20 runs (vs. 7/20 for Instrumentation).

Among the 16 bugs triggered by both modes, Profiling achieves a shorter or equal mean TTT in all cases—14 show clear improvement and 2 are tied. Speedups range from modest (e.g., `XML009` via `xmllint`: 1.2×) to substantial (e.g., `XML009` via `xml_read_memory`: 5.3×; `SQL002`: 10×). Overall, Profiling achieves a shorter TTT in 193 out of 278 shared bug–run instances (69%). Throughput gains correlate directly with bug discovery improvements. The driver with the largest speedup, sqlite3 (7.1×), exhibits the most dramatic TTT reductions: SQL002 (10× faster), SQL014 (6.2× faster), and SQL018 (3.9× faster). Conversely, where the throughput advantage is smallest, Profiling still matches or outperforms Instrumentation on average. In these cases, individual runs exhibit higher variance, consistent with the expectation that stochastic variation in input generation dominates when the throughput advantage is modest.

> ***Summary (RQ2, bug discovery):*** Removing coverage instrumentation yields 1.3× to 7.1× higher fuzzing throughput, which translates into superior bug finding performance. Profiling achieves shorter or equal mean TTT in all 16 bug cases.
> This result indicates that the *tax* of coverage instrumentation negates its diagnostic value for residual risk assessment; by trading observation granularity for raw execution speed, sampled profiling is more effective for rapid bug discovery.

4.2.2 In-depth analysis of throughput and residual risk on FuzzBench. We take a closer look at how the degree of observability—controlled by the sampling frequency and sampling mode—affects both execution throughput and discovery probability estimation on the FuzzBench subjects. We evaluate relative frequencies f of 0.5, 1, 2, 4, and 8 (i.e., from one sample per two executions to eight samples per execution) and compare the top and stack sampling modes (f=1).

Table 6 shows the average execution throughput and estimated discovery probability ($\hat{GT}$) measured at 24 hours under both Instrumentation and Profiling modes for each subject in FuzzBench. A lower discovery probability is desirable—provided the total number of executions is not smaller—as it means the fuzzer has explored more of the program's behavior and the estimator reflects less residual risk. Throughput decreases monotonically with f across all top configurations, as more frequent sampling increases overhead: the average throughput ratio drops from 2.90× at f=0.5 to 1.30× at f=8. At $f \leq 1$, all seven subjects maintain throughput above Instrumentation; at f=2, only libpcap drops slightly below (0.94×); and at $f \geq 4$, proj4 also falls below, while sqlite3 retains the highest gain throughout (2.14× even at f=8).

As shown in Table 7, the per-subject throughput variation is explained by the *interrupt rate* of the profiler. The profiler fires

Table 6: Average throughput (exec/s) and estimated discovery probability ($\hat{GT}$) after 24 hours. For top sampling, we vary the frequency f from 0.5 to 8; for stack sampling, we fix f at 1.0. Parenthesized values show the ratio to full coverage (Instrumentation). Bold marks the best Profiling configuration per subject.

Subject		Instrumentation	Profiling with a top mode: Profiling / top					Profiling / stack
			f=0.5	f=1	f=2	f=4	f=8	f=1.0
proj4	exec/s	1852	3370 (1.82)	2658 (1.44)	1879 (1.01)	1188 (0.64)	888 (0.48)	**3484 (1.88)**
	$\hat{GT}$	3.2e-7	**6.4e-7 (2.02)**	8.2e-7 (2.61)	1.2e-6 (3.75)	2.0e-6 (6.20)	2.6e-6 (8.35)	9.6e-7 (3.03)
sqlite3	exec/s	1695	**7230 (4.27)**	6690 (3.95)	5870 (3.46)	4828 (2.85)	3621 (2.14)	5574 (3.29)
	$\hat{GT}$	1.7e-6	**5.3e-7 (0.32)**	5.8e-7 (0.35)	7.1e-7 (0.43)	9.4e-7 (0.56)	1.4e-6 (0.82)	8.1e-7 (0.48)
mbedtls	exec/s	2277	**6713 (2.95)**	6284 (2.76)	5627 (2.47)	4770 (2.10)	3751 (1.65)	4872 (2.14)
	$\hat{GT}$	3.5e-8	**2.9e-8 (0.81)**	2.9e-8 (0.83)	3.3e-8 (0.93)	3.1e-8 (0.89)	4.2e-8 (1.19)	3.2e-8 (0.90)
libxml2	exec/s	3183	**12004 (3.77)**	10580 (3.32)	8651 (2.72)	6472 (2.03)	4362 (1.37)	7675 (2.41)
	$\hat{GT}$	9.5e-8	**9.2e-8 (0.96)**	1.0e-7 (1.10)	1.2e-7 (1.23)	1.6e-7 (1.69)	2.3e-7 (2.40)	1.9e-7 (1.96)
freetype2	exec/s	2733	**8041 (2.94)**	7707 (2.82)	6497 (2.38)	5047 (1.85)	3637 (1.33)	5866 (2.15)
	$\hat{GT}$	2.3e-7	1.4e-7 (0.64)	**1.4e-7 (0.61)**	1.5e-7 (0.68)	2.0e-7 (0.88)	2.7e-7 (1.18)	2.1e-6 (9.17)
harfbuzz	exec/s	1528	**3987 (2.61)**	3936 (2.58)	3581 (2.34)	3083 (2.02)	2516 (1.65)	2664 (1.74)
	$\hat{GT}$	1.7e-7	1.9e-7 (1.08)	**1.6e-7 (0.93)**	1.7e-7 (0.98)	1.6e-7 (0.95)	2.1e-7 (1.19)	1.9e-7 (1.07)
libpcap	exec/s	7426	**14322 (1.93)**	8793 (1.18)	6973 (0.94)	5028 (0.68)	3444 (0.46)	7202 (0.97)
	$\hat{GT}$	5.0e-8	**2.4e-8 (0.47)**	4.3e-8 (0.86)	4.7e-8 (0.94)	6.3e-8 (1.26)	8.9e-8 (1.78)	5.6e-8 (1.11)
Avg. ratio	exec/s	–	2.90	2.58	2.19	1.74	1.30	2.08
	$\hat{GT}$	–	0.90	1.04	1.28	1.78	2.42	2.53

Table 7: Profiler overhead per subject at f=0.5.

Subject	Total Rate (kHz)	In-Target Rate (kHz)	In-Target (%)
proj4	19.5	2.4	12
libpcap	11.6	7.2	62
libxml2	7.4	6.1	83
freetype2	5.6	4.6	82
sqlite3	3.6	3.6	98
mbedtls	3.5	3.4	98
harfbuzz	2.1	1.9	96

`instructions:u` events for the entire worker process. Each interrupt incurs overhead regardless of where the instruction pointer falls, but only events where the IP lies within the target binary produce useful coverage samples; events in shared libraries (`libc`) are discarded.

We distinguish the total interrupt rate (the frequency of profiler interruptions)—from the in-target rate (the subset yielding coverage). Overhead scales with the total rate, which at f=0.5 it ranges from 2.1 kHz (harfbuzz) to 19.5 kHz (proj4); this ordering closely tracks throughput degradation across all subjects. The two subjects whose throughput drops below Instrumentation at higher f—proj4 and libpcap—exhibit the highest total rates: proj4 has only 12% in-target events, meaning 88% of interrupts are wasted on shared-library code, while libpcap similarly incurs a high total rate (11.6 kHz) despite a 62% in-target fraction, due to its high baseline throughput (7,426 exec/s) which leads to more frequent interrupts even at low f.

The discovery probability ratio increases with f (averaging from 0.90 at f=0.5 to 2.42 at f=8). A DP ratio below 1 indicates that Profiling achieves lower residual risk than Instrumentation despite observing coverage through sampling alone. At f=0.5, five of seven subjects achieve a DP ratio ≤ 1 (sqlite3 0.32, libpcap 0.47, freetype2 0.64, mbedtls 0.81, libxml2 0.96), confirming that profiler-based sampling at low frequencies matches or improves upon Instrumentation for residual risk estimation. sqlite3 represents the strongest case: its DP ratio remains below 1 across all f values (0.32 to 0.82), meaning that Profiling achieves both higher throughput and lower residual risk than Instrumentation in every configuration. Conversely, libxml2 is notable for the opposite reason: despite maintaining a throughput advantage(3.77× at f=0.5, 1.37× at f=8), its DP ratio rises from 0.96 to to 2.40. This suggests that for libxml2 higher throughput does not fully compensate for the coarser observation inherent in sampling at higher f.

Top vs. Stack. Comparing configurations at f=1, top achieves higher throughput than stack in most subjects (2.58× vs. 2.08× on average), with the exception of proj4 where stack (1.88×) outperforms top (1.44×). Recall that proj4 has only 12% in-target events under top; stack recovers target frames from shared-library interrupts via call-stack unwinding, raising the effective in-target yield and allowing a lower total interrupt rate for the same coverage frequency. Regarding discovery probability (DP), at f=1, top also achieves a lower average ratio than stack: 1.04 vs. 2.53. While stack is comparable on some subjects (sqlite3, mbedtls), it is substantially worse on others (freetype2: 9.17 vs. 0.61; libxml2: 1.96 vs. 1.10). Overall, the richer per-sample coverage from stack unwinding does not compensate for its throughput loss, making top the preferred sampling mode.

Statistical significance. Profiler-based sampling at $f = 0.5$ achieves significantly higher throughput than coverage-instrumentation for all seven subjects, with $\hat{A}_{12} = 1.000$ and $p < 10^{-7}$ under a one-sided Mann-Whitney U test.

Summary (RQ2, in-depth analysis): Our results reveal a fundamental trade-off between sampling granularity, fuzzing throughput, and residual risk estimation.

- As sampling frequency f increases, throughput decreases due to higher overhead and discovery probability rises due to reduced total executions.
- When the fuzzer spends significant time in external shared libraries, the noise of non-target interrupts drives up overhead without improving coverage.
- The leaner top profiling remains the optimal choice, providing the best balance of high-speed execution and reliable risk estimation.

4.3 Stopping Time Decisions

RQ3: *How effective is profiler-based sampling for stopping-time decisions?* We compare the stopping times determined by a profiler-based estimate against coverage-instrumentation based estimate to assess whether the throughput advantage compensates for estimation error.

A stopping time is the first moment at which the residual risk estimate drops below a given discovery-probability threshold ε (Section 2). As long as the estimate does not underestimate the true residual risk, an earlier stopping time is preferable, because it reduces total testing duration. We evaluate a wide range of thresholds across different subjects. As in RQ2, we fix the must-execute analysis to the inter-procedural level and vary the sampling frequency and mode to analyze their impact on the stopping time decision.

In deployment, ε would be chosen a priori from a desired risk tolerance. For empirical evaluation in this paper, we instead choose thresholds *post hoc*: subjects saturate at widely different discovery-probability levels, so no single ε is non-trivial for all of them. We

Table 8: Stopping time in hours under Instrumentation vs. Profiling/top at various discovery probability thresholds. Parenthesized values indicate the ratio of each method's stopping time to that of full coverage (Instrumentation). Bold indicates the lowest ratio. ■ indicates ratio <1.

DP			Profiling / top				
Threshold	Subject	Instrumentation	f=0.5	f=1	f=2	f=4	f=8
10^{-7}	proj4	–	–	–	–	–	–
	mbedtls	9.41	**7.46 (0.79)**	7.52 (0.80)	7.48 (0.79)	7.76 (0.82)	10.96 (1.16)
	libxml2	20.85	21.74 (1.04)	22.68 (1.09)	**21.53 (1.03)**	–	–
	freetype2	–	–	–	–	–	–
	harfbuzz	–	–	–	–	–	–
	libpcap	11.31	**6.68 (0.59)**	10.29 (0.91)	12.97 (1.15)	15.53 (1.37)	20.02 (1.77)
2×10^{-7}	proj4	–	–	–	–	–	–
	mbedtls	4.30	3.82 (0.89)	**3.71 (0.86)**	3.95 (0.92)	4.39 (1.02)	5.53 (1.29)
	libxml2	13.21	**11.54 (0.87)**	12.77 (0.97)	14.22 (1.08)	19.99 (1.51)	23.60 (1.79)
	freetype2	22.52	17.80 (0.79)	**16.38 (0.73)**	18.83 (0.84)	22.52 (1.00)	–
	harfbuzz	20.56	20.78 (1.01)	**19.87 (0.97)**	20.43 (0.99)	20.46 (1.00)	22.07 (1.07)
	libpcap	5.42	**3.30 (0.61)**	5.83 (1.08)	7.28 (1.34)	8.56 (1.58)	11.40 (2.10)
5×10^{-7}	proj4	13.33	–	–	–	–	–
	mbedtls	1.33	1.78 (1.34)	**1.73 (1.29)**	1.86 (1.39)	2.06 (1.54)	2.52 (1.89)
	libxml2	6.10	**4.92 (0.81)**	5.43 (0.89)	6.03 (0.99)	8.28 (1.36)	11.87 (1.95)
	freetype2	11.85	7.28 (0.61)	**6.88 (0.58)**	7.83 (0.66)	9.80 (0.83)	13.39 (1.13)
	harfbuzz	8.94	8.79 (0.98)	**8.09 (0.90)**	8.76 (0.98)	9.59 (1.07)	11.10 (1.24)
	libpcap	1.56	**1.37 (0.88)**	2.30 (1.47)	2.99 (1.92)	3.76 (2.42)	5.51 (3.54)
10^{-6}	proj4	6.33	**14.35 (2.27)**	19.81 (3.13)	–	–	–
	mbedtls	0.64	1.01 (1.57)	**0.97 (1.51)**	1.03 (1.59)	1.01 (1.57)	1.24 (1.92)
	libxml2	3.43	**2.62 (0.76)**	2.80 (0.82)	3.26 (0.95)	4.31 (1.26)	6.13 (1.79)
	freetype2	5.89	3.64 (0.62)	**3.62 (0.61)**	4.08 (0.69)	5.07 (0.86)	7.07 (1.20)
	harfbuzz	4.95	5.07 (1.03)	4.36 (0.88)	**4.32 (0.87)**	5.16 (1.04)	5.85 (1.18)
	libpcap	0.70	**0.71 (1.01)**	1.11 (1.58)	1.46 (2.08)	2.00 (2.84)	2.84 (4.05)
10^{-5}	sqlite3	3.49	2.09 (0.60)	1.97 (0.56)	1.86 (0.53)	**1.86 (0.53)**	2.22 (0.64)

therefore calibrate the thresholds to the estimated discovery probability observed in the 12 h to 24 h range. This is an evaluation device for selecting comparable, non-trivial ε values, not the stopping criterion itself. For six subjects whose discovery probability reaches the 10^{-7}–10^{-6} range by 24 h, we use thresholds 10^{-7}, 2×10^{-7}, 5×10^{-7}, and 10^{-6}. For sqlite3, whose discovery probability remains above 10^{-6} throughout the campaign, is reported separately at 10^{-5}.

4.3.1 Results. Table 8 shows the time at which each configuration first drops below the given discovery probability threshold. In the majority of cases, Profiling at low f (0.5 or 1) reaches the stopping criterion earlier than Instrumentation, while high f erodes this advantage. At f=0.5, 13 out of 20 reachable subject–threshold combinations (65%) have a stopping time ratio below 1, meaning Profiling reaches the stopping criterion earlier than Instrumentation. At f=1, the proportion is the same: 13 of 20 (65%). As f increases, the advantage diminishes—11 of 19 (57%) at f=2, 4 of 18 (22%) at f=4—and at f=8, only 1 of 17 (6%) remains below 1. This mirrors the RQ2 throughput and DP trade-offs: at low f, throughput gain from removing instrumentation outweighs the coarser coverage observation; however, at high f, the profiling overhead erodes the throughput advantage and delays the stopping time.

A second pattern emerges across thresholds: Profiling performs better at stricter thresholds. At 10^{-7}, 6 of 13 reachable entries (46%) have ratio below 1, while at 10^{-6}—the most lenient threshold for the main subjects—only 9 of 27 (33%) do. This occurs, because stricter thresholds require longer campaigns, providing a dual advantage for sampled profiling: (1) the estimation accuracy improves over time (as shown in RQ1, Figure 4 right, and predicted by the model in Section 3.1) and (2) the throughput gain from removing instrumentation translates into an earlier stopping time. Conversely, at lenient thresholds, stopping occurs so early in the campaign, that the profiler-based estimate is less accurate, reducing the relative benefit of the sampling approach.

sqlite3 is reported separately at 10^{-5} because its discovery probability remains above 10^{-6} throughout the 24 h campaign. At this threshold, all five Profiling configurations reach the stopping criterion faster than Instrumentation, with similar ratios ranging from 0.53× (f=4) to 0.64× (f=8). This demonstrates how execution speed gain from sampling (4.27× at f=0.5 in RQ2) translates to earlier high-confidence stopping .

Statistical significance. At f=0.5, 11 out of 13 reachable entries where Profiling reaches the stopping criterion earlier than Instrumentation are statistically significant ($p < 0.05$ with the one-sided Mann-Whitney U test) with large difference in effect size ($\hat{A}_{12} \geq 0.71$) for 10 of them.

Summary (RQ3, time-to-stop): At low frequencies, a sampling-profiler reaches the stopping criterion earlier than coverage instrumentation in the majority of cases (13 out of 20 at f=0.5). This benefit is more pronounced at stricter thresholds; once the profiler's estimation converges, the raw throughput gain from sampled-profiling translates accelerated stopping decisions.

5 Threats to Validity

Internal validity. Fuzzing is inherently stochastic, so measured differences could arise by chance. To follow recommended evaluation practice [21, 30], we run twenty repetitions per configuration, use 24-hour campaigns, and confirm significance with Mann–Whitney U tests. Residual variance may nonetheless remain for high-variability subjects such as sqlite3.

External validity. Our evaluation covers seven FuzzBench subjects and three Magma subjects, spanning cryptography, parsing, rendering, and networking domains. These subjects are widely used in the fuzzing literature, yet they may not represent all software types; programs with highly irregular control flow or deeply nested state machines, for example, may exhibit different profiling characteristics.

Construct validity. To isolate the effect of the observation mechanism, we disable coverage guidance for input selection in both Instrumentation and Profiling modes, so the fuzzer operates in blackbox mode in both cases. This ensures a fair comparison for residual risk estimation. Rather than a dedicated black-box fuzzer, we use a modified AFL++ in this black-box mode (Section 3.3) to reuse the prior residual risk analysis framework [22] and its integration with FuzzBench and Magma. Because a black-box fuzzer is defined solely by drawing each input independently of the observed coverage, disabling the feedback loop makes AFL++ equivalent in input distribution to a purpose-built black-box fuzzer, while retaining a well-tested, high-throughput execution engine. In Section 7, we discuss the potential of leveraging profiler-based sampling for the purpose of coverage-guided input selection.

6 Related Work

Residual risk estimation and test adequacy. Böhme introduced the concept of software testing as species discovery (STADS) [7], applying the Good-Turing estimator [15] to estimate the probability that the next test input discovers new program behavior.

Subsequent work formalized residual risk estimation for greybox fuzzing [8, 9], and Liyanage et al. [25] extended this to extrapolate coverage growth rates. Lee and Böhme [22] proposed a dependency-aware estimator that accounts for co-occurrence among coverage elements within the same test execution, yielding tighter bounds. All of these approaches assume coverage instrumentation as the observation mechanism. Our work relaxes this assumption, showing that profiler-based sampling with must-execute analysis can provide estimates within one order of magnitude of the instrumentation-based ground truth. Pavlopoulou and Young [28] studied residual test coverage monitoring in deployed software, noting that coverage collection must impose minimal overhead to be accepted by users—a constraint our profiler-based approach directly addresses.

Reducing coverage instrumentation overhead. Coverage instrumentation imposes substantial overhead: source-level instrumentation adds 60–70% latency [11], while binary-only approaches incur far higher costs—QEMU-based dynamic translation suffers 5–7× slowdown [27], and dynamic binary instrumentation frameworks such as DynamoRIO or Intel PIN can impose 50–100× slowdowns [10]. Several lines of work aim to reduce this overhead. INSTRIM [18] exploits control-flow dominance relations to prune redundant instrumentation points. Full-Speed Fuzzing [27] runs an uninstrumented binary by default and activates tracing only when new coverage is detected. Hardware-assisted approaches such as kAFL [31] and SNAP [11] leverage Intel Processor Trace or custom hardware to collect coverage with lower overhead. These techniques reduce but do not eliminate instrumentation; our approach removes it entirely, relying on profiler-based sampling instead, agnostic to the particular fuzzer used.

Fuzzing in coverage-limited environments. A related line of work targets settings where traditional coverage feedback is unavailable or meaningless, most notably embedded firmware running on dedicated hardware, where source instrumentation is infeasible. μAFL [23] recovers coverage non-intrusively from an ARM ETM instruction trace via a debug dongle, and debug-interface fuzzing [13] extracts partial coverage from uninstrumented binaries using the hardware breakpoints of an on-chip debug interface. These systems tackle the *fuzzing* problem: recovering a coverage-guidance signal through specialized hardware so that a coverage-guided fuzzer can operate at all in such environments. Our concern is orthogonal. We do not guide the fuzzer but perform *residual risk analysis*, estimating testing adequacy from the available observations; we therefore rely on commodity profiler sampling rather than dedicated tracing hardware, with must-execute analysis recovering accuracy from sparse samples. Extending this estimation to hardware-constrained settings is a natural application of our approach.

Sampling-based program monitoring. Liblit et al. [24] pioneered remote program sampling for bug isolation, collecting sparse boolean predicates from deployed software to localize faults with minimal overhead. Ren et al. [29] described Google-Wide Profiling, a continuous profiling infrastructure that samples hardware performance counters across data-center workloads for performance analysis. These systems demonstrate that sampling-based observation is practical at scale, but their goals—debugging and performance optimization—differ from ours. We repurpose the same profiling infrastructure (e.g., Linux `perf`) for a new application: residual risk estimation during fuzzing, and introduce must-execute analysis to compensate for the incomplete observations inherent to sampling.

Fuzzing throughput optimization. Beyond reducing instrumentation overhead, several approaches optimize fuzzing throughput through execution mechanics. SnapFuzz [5] accelerates network application fuzzing via lightweight snapshots. AFL++ [14] combines persistent mode, fork server optimizations, and improved scheduling to maximize executions per second. These techniques are complementary to our approach: they optimize how the fuzzer executes inputs, whereas we optimize how program behavior is observed, and the two can be combined.

7 Conclusion

Our evaluation across nineteen real-world subjects demonstrates that, in black-box fuzzing, coverage instrumentation is a *tax* on throughput that frequently negates its own diagnostic utility for residual risk assessment. By eliminating the overhead of coverage instrumentation, our approach achieves up to a 5.2× throughput increase, which translates directly into superior bug-finding speed across all evaluated cases. Crucially, we prove that this gain in efficiency does not come at the cost of statistical rigor: our sampling approach is inherently conservative—overestimating rather than underestimating residual risk—and by employing inter-procedural must-execute analysis, we reduce relative estimation error to 1.73. Finally, we show that this combination of speed and accuracy can enable practitioners to reach confident stopping-time decisions earlier than a coverage instrumentation-based method in the majority of campaigns. Ultimately, these results suggest a paradigm shift for black-box fuzzing: if coverage data were to be used to quantify residual risk, exhaustive instrumentation is not only unnecessary but counter-productive—it can be safely and effectively substituted with the inherently cheaper signal from a sampled profiler.

Our results show that profiler-based sampling is a robust alternative to coverage instrumentation for assessing black-box fuzzing campaigns. By delivering conservative risk estimates, throughput gains from reduced overhead compensates for coarser observation without leading to dangerous over-confidence (i.e., underestimating residual risk). This efficiency makes sampled-profiling a valuable substitute for high-confidence stopping decisions and resource-constrained environments, where traditional coverage tracking is expensive or undesirable.

Not feeding coverage back into the fuzzer entails a trade-off: black-box search discovers new coverage less efficiently than coverage-guided fuzzing, but it achieves higher execution throughput by avoiding the overhead of maintaining coverage-guided feedback. This loss in search efficiency, however, points to a natural extension. The same synergy between sampled profiling and must-execute analysis that we use for risk estimation could also recover a guidance signal: by augmenting coarse samples with inter-procedural analysis, a fuzzer could reconstruct high-fidelity coverage feedback without the *tax* of full instrumentation. Such a hybrid design offers a path toward lightweight coverage-guided fuzzing that retains black-box throughput while amplifying sparse feedback through control-flow analysis.

8 Data Availability Statement

Our replication package, including source code and experimental data, is archived on Zenodo under DOI: https://doi.org/10.5281/zenodo.19244795.

Acknowledgments

We thank the anonymous reviewers for their valuable comments and helpful suggestions. This work is supported by the National Science Foundation under grant numbers 2426162, 2106838, and 2106404. This research is also partially funded by the European Union (ERC project AT_SCALE, 101179366). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them. This work is also supported in part by funding from Amazon, Fetch AI, and Samsung.

References

[1] GitLab 2026. *Aki Helin / Radamsa · GitLab*. GitLab. https://gitlab.com/akihe/radamsa
[2] [n. d.]. *Peach Fuzzer*. https://peachtech.gitlab.io/peach-fuzzer-community/
[3] [n. d.]. *Perf(1) - Linux Manual Page*. https://man7.org/linux/man-pages/man1/perf.1.html
[4] [n. d.]. *Zzuf – Caca Labs*. http://caca.zoy.org/wiki/zzuf
[5] Anastasios Andronidis and Cristian Cadar. 2022. SnapFuzz: High-Throughput Fuzzing of Network Applications. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2022-07-18) *(ISSTA 2022)*. Association for Computing Machinery, 340–351. doi:10.1145/3533767.3534376
[6] Georgios Bitzes and Andrzej Nowak. 2014. The Overhead of Profiling Using PMU Hardware Counters. (2014), 1–16. doi:10.5281/zenodo.10800
[7] Marcel Böhme. 2018. STADS: Software Testing as Species Discovery. 27, 2 (2018), 7:1–7:52. doi:10.1145/3210309
[8] Marcel Böhme. 2022. Statistical Reasoning about Programs. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results* (New York, NY, USA, 2022-10-17) *(ICSE-NIER '22)*. Association for Computing Machinery, 76–80. doi:10.1145/3510455.3512796
[9] Marcel Böhme, Danushka Liyanage, and Valentin Wüstholz. 2021. Estimating Residual Risk in Greybox Fuzzing. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (New York, NY, USA, 2021) *(ESEC/FSE 2021)*. Association for Computing Machinery, 230–241. doi:10.1145/3468264.3468570
[10] Daniele Cono D'Elia, Emilio Coppa, Simone Nicchi, Federico Palmaro, and Lorenzo Cavallaro. 2019. SoK: Using Dynamic Binary Instrumentation for Security (And How You May Get Caught Red Handed). In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security* (New York, NY, USA, 2019-07-02) *(Asia CCS '19)*. Association for Computing Machinery, 15–27. doi:10.1145/3321705.3329819
[11] Ren Ding, Yonghae Kim, Fan Sang, Wen Xu, Gururaj Saileshwar, and Taesoo Kim. 2021. Hardware Support to Improve Fuzzing Performance and Precision. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2021-11-13) *(CCS '21)*. Association for Computing Machinery, 2214–2228. doi:10.1145/3460120.3484573
[12] Joao Mario Domingos, Pedro Tomas, and Leonel Sousa. 2021. *Supporting RISC-V Performance Counters through Performance Analysis Tools for Linux (Perf)*. arXiv:2112.11767 [cs] doi:10.48550/arXiv.2112.11767
[13] Max Eisele, Daniel Ebert, Christopher Huth, and Andreas Zeller. 2023. Fuzzing Embedded Systems Using Debug Interfaces. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2023-07-13) *(ISSTA 2023)*. Association for Computing Machinery, 1031–1042. doi:10.1145/3597926.3598115
[14] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. https://www.usenix.org/conference/woot20/presentation/fioraldi
[15] I. J. Good. 1953. The Population Frequencies of Species and the Estimation of Population Parameters. 40, 3/4 (1953), 237–264. jstor:2333344 doi:10.1093/biomet/40.3-4.237
[16] Brendan Gregg. 2016. The Flame Graph. 59, 6 (2016), 48–57. doi:10.1145/2909476
[17] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. 4, 3 (2020), 1–29. doi:10.1145/3428334
[18] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. 2018. INSTRIM: Lightweight Instrumentation for Coverage-guided Fuzzing. In *Proceedings 2018 Workshop on Binary Analysis Research* (San Diego, CA, 2018). Internet Society. doi:10.14722/bar.2018.23014
[19] Intel Corporation. 2026. *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
[20] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a Warehouse-Scale Computer. 43 (2015), 158–169. Issue 3S. doi:10.1145/2872887.2750392
[21] 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* (New York, NY, USA, 2018-10-15) *(CCS '18)*. Association for Computing Machinery, 2123–2138. doi:10.1145/3243734.3243804
[22] Seongmin Lee and Marcel Böhme. 2026. Dependency-Aware Residual Risk Analysis. In *Proceedings of the 48th IEEE/ACM International Conference on Software Engineering* (2026) *(ICSE'26)*.
[23] Wenqiang Li, Jiameng Shi, Fengjun Li, Jingqiang Lin, Wei Wang, and Le Guan. 2022. µAFL: Non-Intrusive Feedback-Driven Fuzzing for Microcontroller Firmware. In *Proceedings of the 44th International Conference on Software Engineering* (New York, NY, USA, 2022-07-05) *(ICSE '22)*. Association for Computing Machinery, 1–12. doi:10.1145/3510003.3510208
[24] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. 2003. Bug Isolation via Remote Program Sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation* (New York, NY, USA, 2003) *(PLDI '03)*. Association for Computing Machinery, 141–154. doi:10.1145/781131.781148
[25] Danushka Liyanage, Seongmin Lee, Chakkrit Tantithamthavorn, and Marcel Böhme. 2024. Extrapolating Coverage Rate in Greybox Fuzzing. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE'24)* (New York, NY, USA, 2024-04-12) *(ICSE '24)*. Association for Computing Machinery, 1–12. doi:10.1145/3597503.3639198
[26] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. 2021. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (New York, NY, USA, 2021-08-18) *(ESEC/FSE 2021)*. Association for Computing Machinery, 1393–1403. doi:10.1145/3468264.3473932
[27] Stefan Nagy and Matthew Hicks. 2019. Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing. In *2019 IEEE Symposium on Security and Privacy (SP)* (2019-05). 787–802. doi:10.1109/SP.2019.00069
[28] C. Pavlopoulou and M. Young. 1999. Residual Test Coverage Monitoring. In *Proceedings of the 1999 International Conference on Software Engineering* (1999-05). 277–284. doi:10.1145/302405.302637
[29] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. 2010. Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers. 30, 4 (2010), 65–79. doi:10.1109/MM.2010.68
[30] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. 2024. SoK: Prudent Evaluation Practices for Fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)* (2024-05-19). 1974–1993. arXiv:2405.10220 [cs.SE] doi:10.1109/SP54263.2024.00137
[31] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. 167–182. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo
[32] András Vargha and Harold D. Delaney. 2000. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. 25, 2 (2000), 101–132. doi:10.3102/10769986025002101