Poster: AntiBench: The Benchmarking Framework of Anti-Fuzzing Techniques

Keigo Yoshioka, Shogo Konishi, Naomasa Matsubayashi, Yuichi Sugiyama
Ricerca Security, Inc.
Email: {keigoy, shogok, naomasam, yuichis}@ricsec.co.jp

***Abstract*—Recently, fuzzing techniques have been leveraged by malicious attackers to explore zero-day vulnerabilities, posing a serious security threat. To resolve this, researchers have proposed anti-fuzzing techniques, defensive mechanisms embedded in publicly distributed binaries. However, previous studies employ heterogeneous evaluation metrics and experimental setups, which makes uniform and reproducible evaluation difficult. This research addresses this challenge by proposing AntiBench, a benchmarking framework that enables quantitative and consistent evaluation. In AntiBench, we present the systematic classification of anti-fuzzing techniques and standardized evaluation criteria. As part of our preliminary evaluation, we implemented several anti-fuzzing techniques based on our classification method and demonstrated the effectiveness of our benchmarking design.**

I. Introduction

Fuzzing is a widely adopted technique that automatically detects vulnerabilities by repeatedly executing a target program with randomly generated inputs, and it has contributed to the discovery of numerous vulnerabilities. However, such techniques can be misused by malicious attackers to identify previously unknown vulnerabilities in publicly distributed binaries, posing a serious threat by realizing zero-day attacks.

Recently, to protect software against malicious fuzzing, researchers have proposed anti-fuzzing as a defensive mechanism. Anti-fuzzing refers to a set of techniques that harness the operational principles and implementation features of fuzzers to degrade their vulnerability discovery capabilities.

Despite the growing body of anti-fuzzing research, a uniform and reliable evaluation of it remains challenging. This is because existing studies adopt inconsistent experimental setups, evaluation metrics, and benchmark targets; thus, we are unable to evaluate each strategy under uniform criteria or observe its effect across various settings.

To address these challenges, we propose a systematic classification of anti-fuzzing techniques and introduce **AntiBench**, a benchmarking platform based on this classification. AntiBench decomposes techniques into core building blocks, which we refer to as *primitives*. This modular design enables flexible composition and comparison among different techniques.

II. Background and Motivation

Anti-fuzzing techniques are designed to degrade fuzzer effectiveness while minimizing performance overhead for benign users. Therefore, they must satisfy the following two design goals: **G1**, degrading fuzzers' abilities to discover zero-day vulnerabilities, and **G2**, minimizing runtime overhead during normal execution.

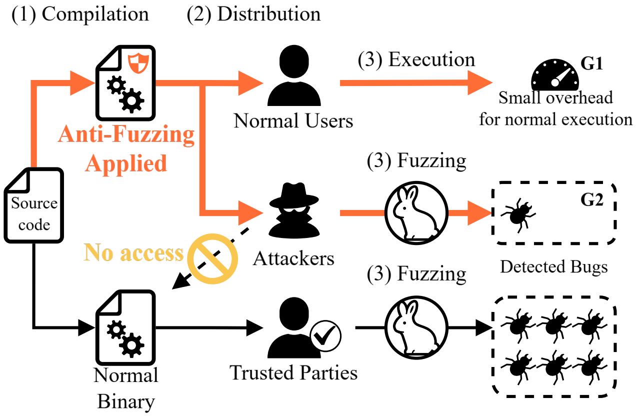


Fig. 1. Workflow of anti-fuzzing and its defensive role

To achieve these goals, anti-fuzzing techniques must incorporate the following two fundamental strategies. The first is the **hindrance strategy**, which dictates how the fuzzer's operations are hindered. The second is the **insertion strategy**, which determines the specific locations within the binary where the hindrance code snippet is instrumented. The hindrance strategy primarily contributes to achieving **G1**, while the insertion strategy is responsible for **G2**. These two strategies exist in an inherent trade-off; prioritizing one often requires a compromise in the effectiveness of the other.

Hence, evaluating anti-fuzzing techniques requires measuring the degree to which both **G1** and **G2** are achieved. However, existing methods lack a standardized benchmark capable of facilitating such a systematic evaluation. The lack of consistency in target programs and fuzzers across previous studies results in their incomparable evaluation results.

Furthermore, previous studies evaluate anti-fuzzing techniques monolithically, in which hindrance and insertion strategies are coupled and assessed together. This coupling obscures the individual contribution of each strategy and prevents standardized comparison under unified conditions.

III. AntiBench: Benchmarking Design

To address these evaluation challenges, we design AntiBench, a benchmarking framework that decomposes anti-fuzzing techniques into design-level primitives and evaluates them under unified criteria.

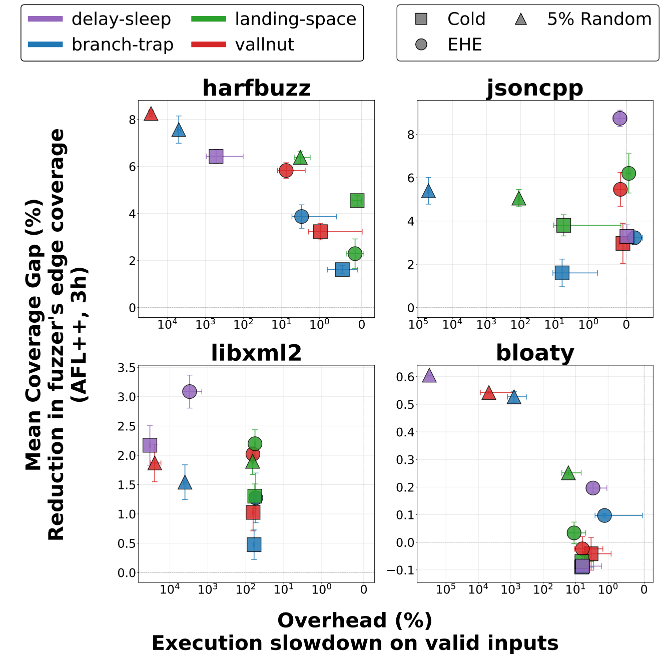


Fig. 2. Results: trade-off between fuzzer hindrance and runtime overhead

TABLE I
ANTI-FUZZING PRIMITIVES USED IN EVALUATION

Strategy	Primitives
Hindrance	BranchTrap (Fuzzification [3]), Delay Sleep (Anti-Fuzz [2]), Landing Space (No-Fuzz [5]), VALL-NUT Landing Space [4]
Insertion	Random Sampling (Baseline), Cold Path (Fuzzification [3]), Error-Handling Path (VALL-NUT [4])

AntiBench systematically classifies anti-fuzzing techniques along two strategies. The hindrance strategy, which targets **G1**, can be categorized into the following two classes. *Feedback hindrance*, which misleads input generation by falsifying feedback information collected by fuzzers, and *execution hindrance*, which degrades fuzzer throughput by deliberately delaying program execution.

The insertion strategies, designed to achieve **G2**, can be classified into two main classes. One method is to apply anti-fuzzing logic to *cold paths*, which are rarely executed during normal execution but are frequently exercised by fuzzers. The other method is to run anti-fuzzing code snippets if and only if the binary is detected to be run by fuzzers.

Based on this design, AntiBench adopts code coverage as the evaluation metric for **G1** and runtime overhead during normal execution as the evaluation metric for **G2**. This enables evaluating **G1** and **G2** independently, also making their inherent trade-off measurable.

IV. PRELIMINARY EVALUATION

To validate the soundness of AntiBench, we re-implemented some existing techniques and conducted a preliminary evaluation based on the following three research questions (RQs).

RQ1 Can AntiBench capture performance variations?

RQ2 Can AntiBench capture trade-off between **G1** and **G2**?

RQ3 Are performance trends of fuzzers reproducible across target binaries?

In our experiments, we evaluated the anti-fuzzing primitives cataloged in Table I. These were integrated by combining selected insertion and hindrance strategies, which were then applied to the four targets (Figure 2.)

We evaluated each instrumented binary using AFL++ [1] for a duration of 3 hours. Figure 2 illustrates the evaluation results based on the metrics defined in Section III. In this figure, the horizontal axis denotes the runtime overhead during normal execution, while the vertical axis represents the effectiveness of the disruption on fuzzers’ coverage discovery capabilities.

RQ1: Cold Path and Error-Handling Path (EHE) strategies incur 100 to 1000 times less overhead than the Random. This demonstrates that AntiBench successfully captures the distinct performance characteristics in different insertion strategies.

RQ2: There is a visible trend from the top-left to the bottom-right in each plot, indicating a weak negative correlation between fuzzer hindrance and overhead. This verifies that AntiBench successfully captures the fundamental trade-off that increased coverage disruption correlates with higher runtime overhead.

RQ3: The combination of EHE and Landing Space consistently achieves near-optimal performance in `jsoncpp`. We also observed similar trends across all other binaries, validating the reproducibility of our evaluation.

V. CONCLUSION AND FUTURE DIRECTION

We address the lack of standardized evaluation for anti-fuzzing techniques by introducing AntiBench, a benchmarking framework based on two design goals of anti-fuzzing. It enables systematic comparison under unified metrics. Our preliminary evaluation demonstrates that AntiBench captures distinct effects of different strategies and validates the effectiveness of our benchmarking methodology.

However, our preliminary evaluation is based on a limited set of primitives and metrics. To evaluate the impact on fuzzers more accurately, future work will extend the benchmark to a broader range of existing techniques and target programs, and investigate interactions among multiple primitives as well as the effect of different fuzzer configurations.

ACKNOWLEDGMENT

This paper is based on results obtained from a project, JPNP24003, commissioned by the New Energy and Industrial Technology Development Organization (NEDO).

REFERENCES

[1] A. Fioraldi et al. AFL++: Combining incremental steps of fuzzing research. In *USENIX Conference on Offensive Technologies*, 2020.

[2] E. Güler et al. ANTIFUZZ: Impeding fuzzing audits of binary executables. In *USENIX Conference on Security Symposium*, 2019.

[3] J. Jung et al. FUZZIFICATION: Anti-fuzzing techniques. In *USENIX Conference on Security Symposium*, 2019.

[4] Y. Li et al. VALL-NUT: Principled anti-grey box fuzzing. In *International Symposium on Software Reliability Engineering*, 2021.

[5] Z. Zhou et al. No-Fuzz: Efficient anti-fuzzing techniques. In *Security and Privacy in Communication Networks*, 2022.

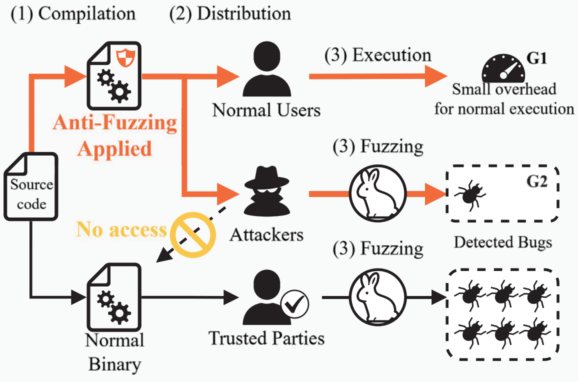
Poster: AntiBench: The Benchmarking Framework of Anti-Fuzzing Techniques

Keigo Yoshioka, Shogo Konishi, Naomasa Matsubayashi, Yuichi Sugiyama
Ricerca Security, Inc.

1. Anti-Fuzzing Goals

Fuzzing: Automated bug finding

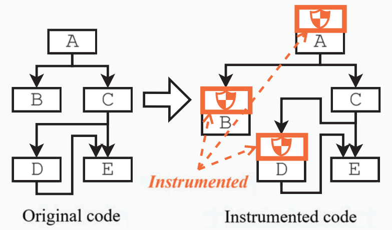
- Defenders: 10K+ bugs found (OSS-Fuzz)
- Attackers: 0-day hunting tool



Design Goals:
G1: Low fuzzing efficiency (for attackers)
G2: Low normal overhead (for users)

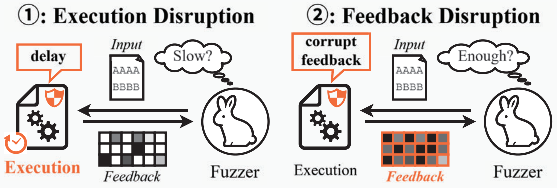
2. Two Dimensions of Anti-Fuzzing

D1: How to hinder fuzzers



When inserting D1 codes...
- More codes →**G2** overhead
- Less codes → weak **G1**

Insertion and D1 requires **trade-off**

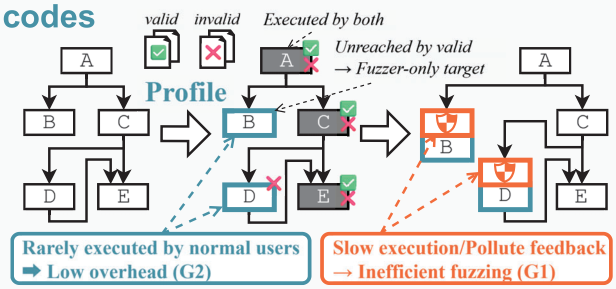


Existing Method	Hinderance Mechanism
delay-sleep [1]	Injecting short delay (`sleep` calls)
branch-trap [2]	Fabricating fake `if` branches
landing-space [3]	Redirecting to dummy blocks
vallnut [4]	Redirecting to large dummy blocks

D2: Where to insert D1 codes

Existing Method	Description
Cold [1][2][3]	Edges never executed by valid inputs: `exec(valid) = 0`

EHE [4]	Error-handling edges:
	`exec(valid) = 0 ∧`
	`exec(invalid) > 0`



Rarely executed by normal users
➡ Low overhead (G2)

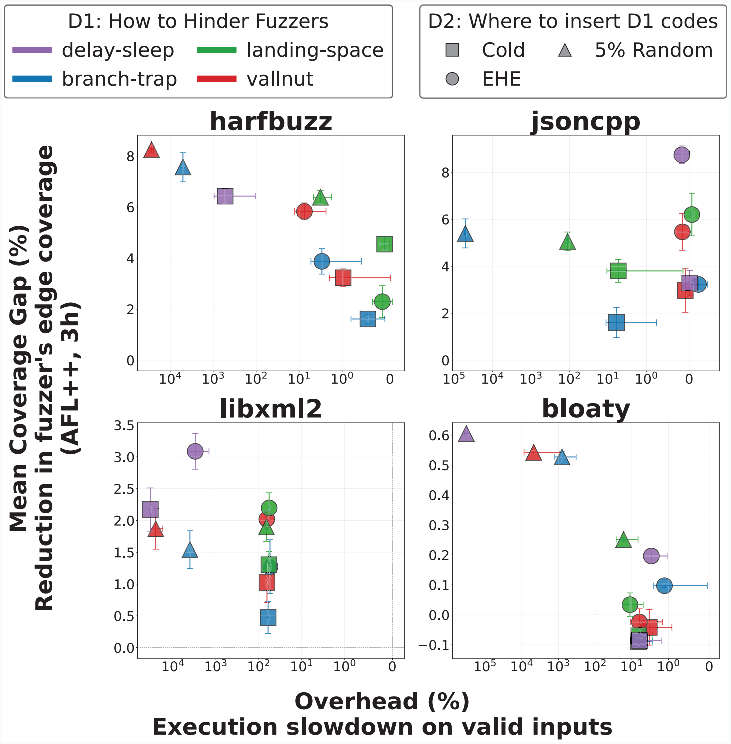
Slow execution/Pollute feedback
→ Inefficient fuzzing (G1)

3. AntiBench: Motivation and Benchmarking Design

Evaluation Challenge: D1 & D2 combined and evaluated together
→ Unclear contributions of each dimension
→ No benchmark for isolating methods & dimensions
🎯 Independently evaluate D1/D2 & their trade-off

Our Proposal: AntiBench
1. Classification→ categorize techniques & allow combinations
2. Standardized Criteria (to evaluate D1/D2 separately)
→ **G1**: coverage reduction & **G2**: overhead of normal execution

4. Preliminary Evaluation



RQ1: Can AntiBench capture performance variations?
→ Cold & EHE: 100-1000x less overhead than Random
→ Successfully captured differences among D2

RQ2: Can AntiBench capture trade-off b/w G1 & G2?
→ Visible trend from top-left to bottom-right
∴ Captured the trend: the larger gap, the more overhead

RQ3: Performance trends reproducible across targets?
→ EHE x Landing Space: near-optimal across targets

5. Conclusion & Future Work

✓ Proposed systematic classification (D1 × D2)
✓ Evaluated previous methods with uniformed metrics
✓ Validated benchmarking methods through evaluation
□ Expand evaluation (more methods & combinations)
□ Design new anti-fuzzing techniques using AntiBench

References:
[1] E. Güler et al., "AntiFuzz: Impeding Fuzzing Audits of Binary Executables," USENIX Security '19
[2] J. Jung et al., "Fuzzification: Anti-Fuzzing Techniques," USENIX Security '19
[3] Z. Zhou et al., "No-Fuzz: Efficient Anti-Fuzzing Techniques," SecureComm '22
[4] Y. Li et al., "VALL-NUT: Principled Anti-Grey Box Fuzzing," ISSRE '21

K Program
Network and Distributed System Security (NDSS 2026)
Supported by NEDO (JPNP24003)