Counter-Fuzzing: Stealthy and Effective Techniques Against Coverage-Guided Fuzzers

Chia-Yu Chang[*], Hsiang-Jou Chuang[*], Hyungmin Kim[*], Sheng-Fang Chien[†]
[*]Department of Computer Science, Virginia Tech, Alexandria, Virginia, USA
chienccc@vt.edu, jou11@vt.edu, henrykim9319@vt.edu
[†]Department of Electrical and Computer Engineering, Virginia Tech, Alexandria, Virginia, USA
serenechien@vt.edu

***Abstract*—Coverage-guided fuzzing is a powerful tool for automated vulnerability discovery, but its efficiency opens the door to *counter-fuzzing*—deliberate program modifications that disrupt fuzzing. We systematically evaluate three such techniques—busy loop (BL), input-dependent delay (IDD), and dynamic self-modification (DSM)—embedded in the `cJSON` parser. In 60-minute AFL++ tests, BL significantly degrades performance but is easily detected, IDD stealthily slows execution while slightly improving coverage, and DSM nearly halts fuzzing altogether. These results highlight a trade-off between disruption and stealth, offering practical insights for both fuzzing tool designers and defenders.**

***Index Terms*—counterfuzzing, fuzz testing, anti-analysis, program instrumentation, software security.**

I. Introduction

Coverage-guided fuzzing has become a cornerstone of automated vulnerability discovery in both academia and industry. Iconic tools such as *American Fuzzy Lop* (AFL) [1] and *libFuzzer* [2] repeatedly mutate inputs, execute the target program, and measure edge coverage to guide exploration of new code paths. Large-scale services like Google's *OSS-Fuzz* have leveraged these techniques to report more than 10 500 high-severity bugs across more than 700 open-source projects since 2016 [3]. As fuzzers permeate secure development lifecycles, software vendors have begun to explore *counter-fuzzing*: deliberate program modifications that hamper black-box fuzzing attacks.

Problem. Existing research has proposed isolated counter-fuzzing ideas—e.g., algorithmic slow-downs [4] or feedback spoofing [5]—but their practical impact has never been *systematically quantified*. In particular, the trade-offs among three essential dimensions remain unclear:

(i) **Throughput degradation** (execs s^{-1}),
(ii) **Code-coverage impact**, and
(iii) **Detectability/stealth**.

A defender needs concrete data to choose between maximum disruption and minimal visibility.

Our Approach. We select the widely deployed `cJSON` parser (a baseline benchmark in FuzzBench [7]) and implement three representative counterfuzzing gadgets:

(a) **Busy Loop (BL)** – inserts a constant spin loop inside `parse_value`.
(b) **Input-Dependent Delay (IDD)** – triggers long delays only for rare hash patterns.
(c) **Dynamic Self-Modification (DSM)** – overwrites the entry of `cJSON_ParseWithOpts` at runtime, redirecting execution to a slow stub.

II. Background and Related Work

Coverage-guided fuzzing has become a cornerstone of automated vulnerability discovery in both academic and industrial contexts. Tools such as *American Fuzzy Lop* (AFL) [1] and *libFuzzer* [2] have demonstrated their effectiveness by generating mutated inputs and tracking edge coverage to uncover deep software bugs. Google's OSS-Fuzz infrastructure extends this approach at scale, identifying over 10 500 high-severity vulnerabilities across more than 700 open-source projects since 2016 [3].

To better understand the concept of counter-fuzzing and successfully implement our project, we reviewed academic papers and industry practices related to fuzzing techniques. Fuzzing has proven to be one of the most effective strategies for discovering software bugs and security vulnerabilities. A typical fuzzing workflow consists of three main components: test case generation, code coverage tracing, and crash triage. Among these, coverage tracing plays a critical role. It allows the fuzzer to evaluate which inputs are beneficial for exploring new code paths, thereby guiding the generation of future test cases. However, a significant inefficiency exists in the way most coverage-guided fuzzers operate today. These fuzzers trace every single test case, yet in practice, fewer than 1 in 10,000 inputs actually exercise new code and contribute to increased coverage. Despite this low yield, more than 90 percent of the fuzzing time is spent executing and tracing inputs that provide no value. This leads to substantial computational waste and slows down the fuzzing process.

To tackle this issue, researchers introduced the concept of coverage-guided tracing and developed a system called UnTracer. UnTracer is specifically designed to reduce the overhead of tracing by selectively tracing only those test cases that are likely to explore new code paths. This approach significantly improves fuzzing throughput without sacrificing code coverage effectiveness.

To address the inefficiencies inherent in coverage-guided fuzzing, UnTracer introduces a selective tracing mechanism that drastically reduces overhead by only tracing test cases that are likely to increase code coverage. The system consists of three main components: an interest oracle, a trace binary, and forkserver instrumentation.

- **Interest Oracle**: The interest oracle is designed to detect whether a test case reaches a previously unexplored basic block. It achieves this by inserting a SIGTRAP-triggering instruction (0xCC) at the beginning of every uncovered basic block. If an input executes this instruction, a SIGTRAP signal is raised, indicating that the input is coverage-increasing. This oracle is implemented using Dyninst's static control-flow analysis to ensure that instrumentation does not interfere with the correct initialization of the program. Notably, since 0xCC is only one byte, it minimally affects small basic blocks and preserves the original semantics of the code.
- **Trace Binary**: Once a coverage-increasing input is detected, it is forwarded to the trace binary for comprehensive coverage collection. The trace binary is also instrumented using Dyninst, with each basic block containing a callback function that logs execution. Additionally, the binary is equipped with a forkserver to enhance performance by reducing process startup overhead. This component enables precise tracking of all executed blocks and updates the fuzzer's global coverage map accordingly.
- **Forkserver Instrumentation**: To further reduce execution overhead, UnTracer incorporates forkserver instrumentation, a technique inspired by AFL. By leveraging fork() to create child processes from a persistent parent process, the system avoids redundant initialization and accelerates test case evaluation. This mechanism is particularly effective in scaling fuzzing campaigns while maintaining low latency and high throughput.

In response to these increasingly powerful fuzzing tools, researchers have explored *counter-fuzzing* techniques that deliberately disrupt the feedback loop or execution efficiency of fuzzers. For example, *SlowFuzz* [4] targets algorithmic complexity vulnerabilities by inducing worst-case behavior, while *T-Fuzz* [5] spoofs coverage information by modifying program logic. However, prior studies largely focus on individual techniques in isolation and lack a systematic evaluation across critical trade-offs such as throughput, coverage, and stealth.

To address this gap, we implement and evaluate three representative counter-fuzzing strategies within the widely used `cJSON` parser: (1) a busy loop (BL) that inserts a constant spin loop, (2) an input-dependent delay (IDD) that triggers slowdowns only on rare hash patterns, and (3) dynamic self-modification (DSM) that rewrites the program entry point at runtime to redirect execution to a slow stub.

Each variant is benchmarked using AFL++ over a 90 min fuzzing campaign. The baseline achieves **16.57** bitmap bits of coverage at **647 exec** $\mathbf{s^{-1}}$ ($\approx 3.3 \times 10^6$ total executions). BL dramatically degrades performance to **2.66 exec** $\mathbf{s^{-1}}$ and **13.30** bits of coverage with only 1×10^4 total executions. IDD, though stealthier, slows execution to **533 exec** $\mathbf{s^{-1}}$ but surprisingly improves coverage to **17.07**. DSM is the most disruptive, reducing coverage to **0.03** with just 1.2×10^3 executions and a throughput of **11.1 exec** $\mathbf{s^{-1}}$.

These results provide the first quantitative comparison of counter-fuzzing techniques and inform a disruption–stealth decision matrix: DSM maximizes disruption at the cost of stability, IDD balances stealth and effectiveness, while BL is simple but easily detectable.

III. Design of Counter-Fuzzing Techniques

A. High-Level Design Goals

The primary goals for counter-fuzzing are to introduce delays and perturbations into the fuzzing process while preserving the program's functional paths. The design principles for our counter-fuzzing techniques are:

- **Stealth**: The techniques should be subtle enough not to be easily detected by fuzzers or the instrumentation.
- **Generality**: These techniques should work across different fuzzing tools.
- **Effectiveness**: The goal is to slow down the fuzzing process by introducing artificial delays, without affecting the actual path exploration or coverage.

B. Description of Each Technique

*1) **Busy Loop (BL)**:* A busy loop is inserted into key parts of the code to waste CPU cycles, forcing the fuzzer to spend more time executing the code without affecting the paths explored. The key target function is 'parse_value()', the core of the cJSON parser.

Stealth: Low – Easy to detect if instrumentation is used.
Complexity: Low – Simple to implement but effective at slowing down fuzzing.

*2) **Input-Dependent Delay (IDD)**:* This method introduces delays based on the hash of the input data. If the hash of the input matches a specific pattern, a time-consuming function like a busy loop is triggered.

Stealth: Medium – The delay is conditional, making it harder to predict and evade.
Complexity: Medium – Requires adding a hash function and conditional logic.

*3) **Dynamic Self-Modification (DSM)**:* DSM works by modifying the program's binary at runtime. It inserts a jump instruction leading to a time-consuming function, slowing down the execution. This method targets the 'cJSON_ParseWithOpts()' function, the entry point of JSON parsing.

Stealth: High – It modifies the program flow dynamically, making it harder for fuzzers to track.
Complexity: High – Involves manipulating the program's memory during execution.

C. Additional No-Fuzz-Inspired Mechanisms

Passive detection methods like instrumentation detection and execution frequency analysis are employed to assess the effectiveness of the counter-fuzzing techniques in evading fuzzing tools.

IV. Implementation

A. Target Program: cJSON

We selected cJSON as the target program for counter-fuzzing due to its small size and simple structure. With only 1,447 basic blocks, cJSON is an ideal target for fuzzing. The main parsing logic resides in ‘parse_value()‘, making it a convenient entry point for applying our counter-fuzzing techniques.

B. Code Integration Points

- **Busy Loop**: Inserted into parse_value() to slow down the parsing process for all JSON values.

  1) Using Busy Loop:

```
function parse_value()
    if is_json_value()
        busy_loop()
        process_json_value()
    end if
end function

function busy_loop()
    counter = 0
    max_counter = 10000000
    while counter < max_counter
        counter = counter + 1
    end while
end function
```

- **Input-Dependent Delay**: Introduced within parse_value() to trigger delays based on the input data’s hash.

  2) Using Input-Dependent Delay:

```
function parse_value(input)
    if is_json_value(input)
        hash_val =
         ↪  compute_hash(input)
        if hash_val matches specific
         ↪  pattern
            busy_loop()
        end if
        process_json_value(input)
    end if
end function

function compute_hash(data)
    hash = 0
    for each byte in data
        hash = (hash * 31 + byte) %
         ↪  1000000007
    end for
    return hash
end function

function busy_loop()
    counter = 0
    max_counter = 10000000
    while counter < max_counter
        counter = counter + 1
    end while
end function
```

- **Dynamic Self-Modification**: Applied in cJSON_ParseWithOpts() to modify the function pointer, redirecting execution to a slow stub.

3) Using Dynamic Self-Modification:

```
function
 ↪  cJSON_ParseWithOpts(input_data)
    original_function_pointer = get_⌋
     ↪  function_pointer(parse_value)

    // Modify the function pointer to
     ↪  point to a slow version
    set_function_pointer(parse_value,
     ↪  slow_stub)

    result = parse_json(input_data)
     ↪  // Call the modified function

    // After parsing, restore the
     ↪  original function pointer
    set_function_pointer(parse_value,
     ↪  original_function_pointer)

    return result
end function

// Slow stub that adds delay
function slow_stub(input_data)
    busy_loop()  // Call the busy
     ↪  loop for intentional delay
    return parse_value(input_data)
     ↪  // Call the original function
     ↪  after delay
end function

// Function to get the current
 ↪  function pointer
function get_function_pointer(functi⌋
 ↪  on_name)
    return function_pointers[functio⌋
     ↪  n_name]
end function

// Function to set the function
 ↪  pointer to a new address
function set_function_pointer(functi⌋
 ↪  on_name,
 ↪  new_pointer)
    function_pointers[function_name]
     ↪  = new_pointer
```

```
end function
```

V. Evaluation and Results

A. Code Coverage Over Time

Figure 1 presents the number of unique control-flow edges discovered over time for four fuzzing configurations: Baseline, Busy Loop (BL), Input-Dependent Delay (IDD), and Dynamic Self-Modification (DSM).

Baseline and **IDD** demonstrate rapid edge discovery and stabilize near 17 edges. **BL**, in contrast, shows delayed and gradual growth, plateauing around 13 edges due to the constant slowdown in execution. **DSM** results in near-zero progress, indicating that the fuzzer is entirely obstructed.

This suggests a clear stratification of strategies:

- **High coverage**: Baseline, IDD
- **Moderate coverage**: BL
- **No coverage**: DSM

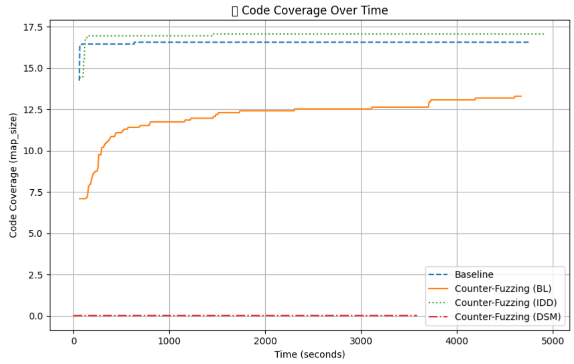


Figure 1. Code coverage (map size) over time. BL slows fuzzing, DSM blocks it entirely.

B. Executions per Second

Execution throughput is a proxy for fuzzing efficiency. Figure 2 shows that **Baseline** initially peaks at over 4000 execs/sec, stabilizing around 650. **IDD** fluctuates widely but maintains moderate throughput. **BL** remains under 10 execs/sec, severely limiting test frequency. **DSM** shows almost complete stagnation.

Figure 3 provides a log-scaled smoothed version, highlighting the orders-of-magnitude difference in fuzzing speed across methods.

C. Quantitative Metrics

To enable standardized evaluation of counter-fuzzing strategies, we define two formal metrics for comparing disruption and stealth:

Disruption Ratio (DR) quantifies throughput degradation relative to the baseline:

$$\mathrm{DR} = 1 - \frac{\mathrm{Exec}_{\mathrm{strategy}}}{\mathrm{Exec}_{\mathrm{baseline}}}$$

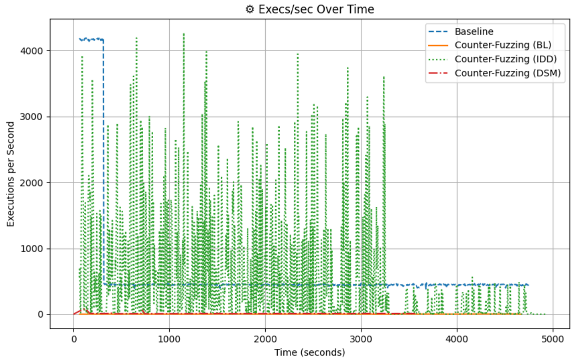


Figure 2. Executions per second across time. DSM halts fuzzing; IDD injects variability.

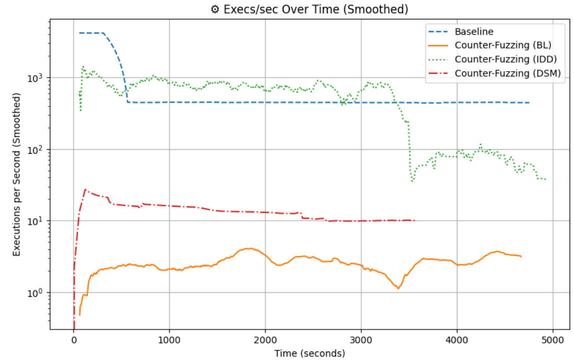


Figure 3. Log-scaled, smoothed execution rate. DSM and BL show low throughput.

where $\mathrm{Exec}_{\mathrm{strategy}}$ is the average executions per second under a counter-fuzzing method, and $\mathrm{Exec}_{\mathrm{baseline}}$ is that of the unmodified baseline.

Stealth Score (SS) captures the temporal stability of execution rates:

$$\mathrm{SS} = \frac{1}{\sigma(\mathrm{Exec/sec})}$$

where σ denotes the standard deviation of execution rate over time. A higher SS indicates more stable (i.e., stealthy) behavior, making detection by fuzzers more difficult.

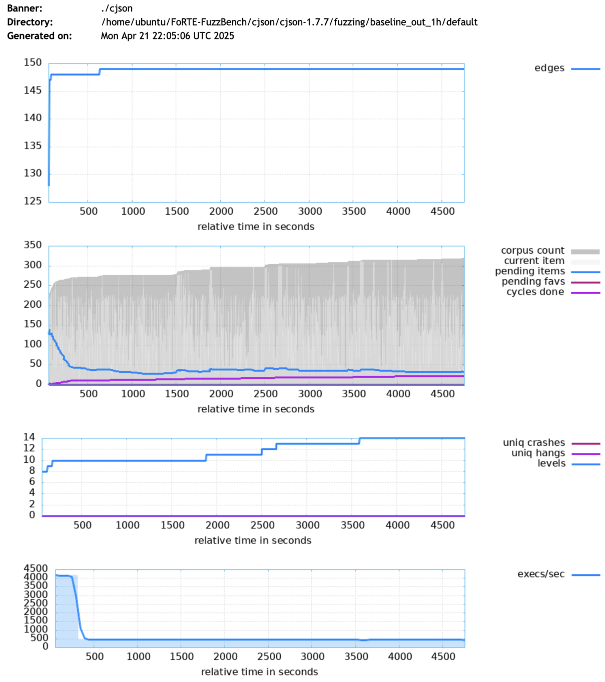
These metrics offer a formal foundation for analyzing the disruption-stealth trade-off that lies at the core of counter-fuzzing.

D. Internal Fuzzer Behavior

Using AFL++'s internal statistics visualizations, we examined runtime behavior:

a) Baseline (Fig. 4): shows healthy fuzzing dynamics—rapid edge discovery, large corpus growth, deep mutation levels, and stable throughput.

b) BL (Fig. 5): yields minimal edge discovery and almost stagnant mutation cycles. The fuzzer is throttled but functional.

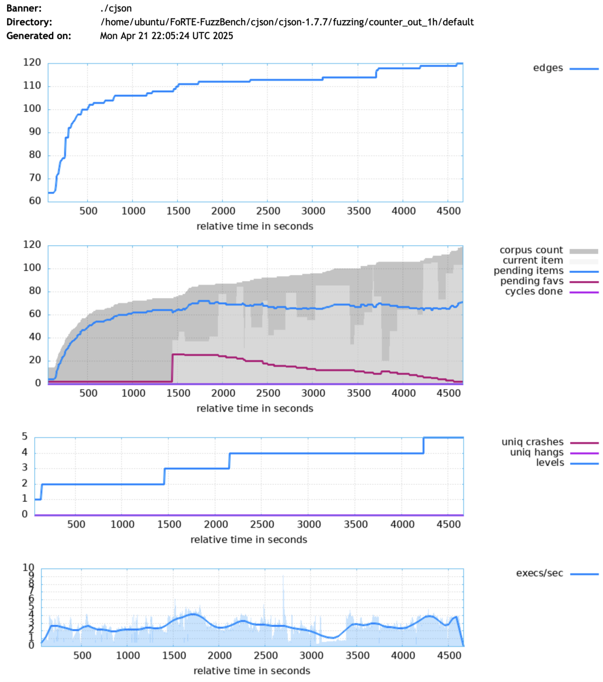


Banner: ./cjson
Directory: /home/ubuntu/FoRTE-FuzzBench/cjson/cjson-1.7.7/fuzzing/baseline_out_1h/default
Generated on: Mon Apr 21 22:05:06 UTC 2025

Figure 4. Baseline: consistent edge discovery, corpus growth, and execs/sec.

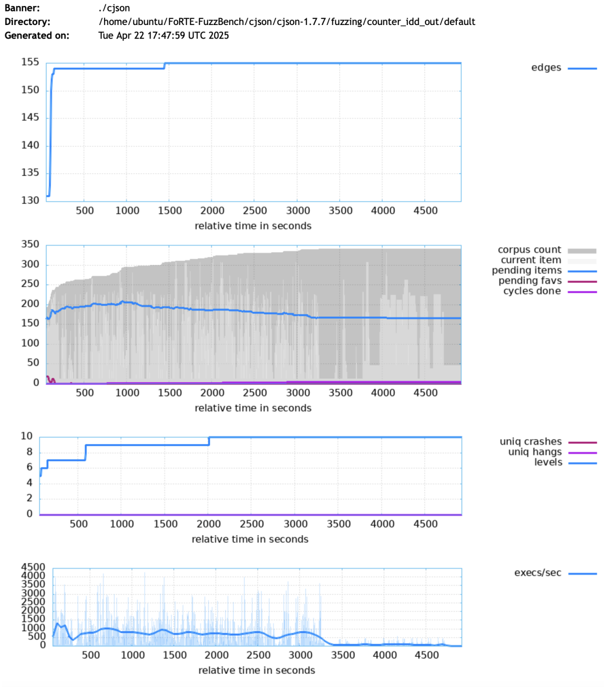
c) IDD (Fig. 6): maintains high exploration but exhibits unstable execution and corpus behavior due to selective delays.

d) DSM (Fig. 7): completely stalls exploration. Mutation, throughput, and edge discovery remain flat.



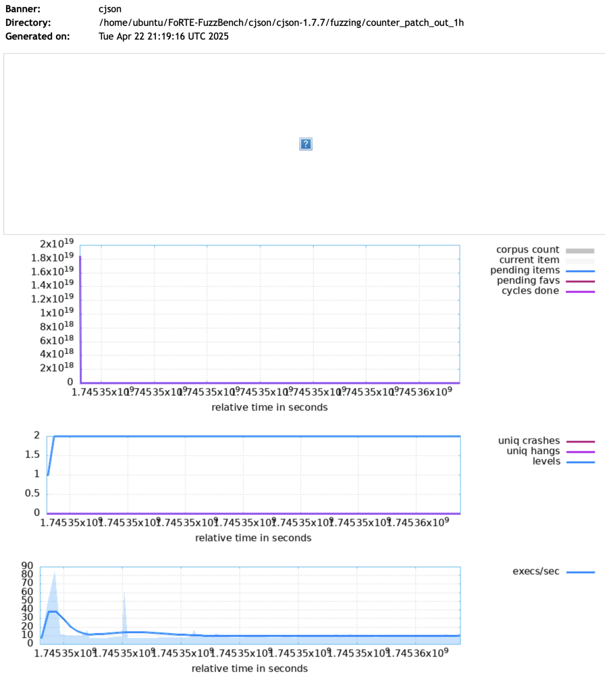
Banner: ./cjson
Directory: /home/ubuntu/FoRTE-FuzzBench/cjson/cjson-1.7.7/fuzzing/counter_out_1h/default
Generated on: Mon Apr 21 22:05:24 UTC 2025

Figure 5. BL: corpus and coverage increase slowly, throughput severely reduced.



Banner: ./cjson
Directory: /home/ubuntu/FoRTE-FuzzBench/cjson/cjson-1.7.7/fuzzing/counter_idd_out/default
Generated on: Tue Apr 22 17:47:59 UTC 2025

Figure 6. IDD: unstable fuzzing but high corpus and edge coverage.



Banner: cjson
Directory: /home/ubuntu/FoRTE-FuzzBench/cjson/cjson-1.7.7/fuzzing/counter_patch_out_1h
Generated on: Tue Apr 22 21:19:16 UTC 2025

Figure 7. DSM: virtually no exploration or execution.

E. Summary of Effectiveness

Table I provides a qualitative overview of how each counter-fuzzing strategy affects key fuzzing metrics: code coverage, execution throughput, and the overall fuzzing state.

The **Baseline** configuration achieves high coverage and execution speed, representing the expected performance of a fully

Table I
COMPARISON OF FUZZING EFFECTIVENESS

Strategy	Coverage	Exec/sec	Fuzzing State
Baseline	High	High	Stable
BL	Low	Very Low	Throttled
IDD	High	Moderate	Noisy
DSM	Very Low	Near-Zero	Blocked

functional fuzzing environment. The BL strategy significantly slows down execution, resulting in a throttled state where the fuzzer progresses but at a much-reduced rate, with lower coverage. IDD maintains high coverage by allowing most inputs to pass normally while selectively delaying a few. This leads to moderate throughput and a noisy but functioning fuzzing state. Lastly, DSM is the most aggressive: it introduces control-flow hijacking that almost completely blocks fuzzing, resulting in very low coverage and near-zero throughput.

This comparison highlights the trade-offs between stealth, effectiveness, and disruption. While BL is easy to implement, it is also the easiest to detect. IDD offers a middle ground, and DSM delivers the strongest disruption with the highest implementation complexity.

VI. DISCUSSION AND ANALYSIS

The evaluation results highlight not only how each technique affects fuzzing outcomes, but also the practical trade-offs involved in deploying them in real systems.

A. Quantitative Summary

Table II consolidates the key experimental results across all strategies. It reports execution volume, code coverage, crash/hang detection, and our proposed metrics—Disruption Ratio (DR) and Stealth Score (SS).

Notably, the Busy Loop (BL) strategy introduces near-complete throughput degradation (DR = 0.9959), while being trivially detectable due to low SS. In contrast, Input-Dependent Delay (IDD) offers high stealth (SS = 0.09) and maintains high coverage, balancing disruption and evasion. DSM causes extreme disruption but has limited stealth, making it most effective but potentially unstable.

These quantitative results validate the trade-off spectrum between coverage, disruption, and stealth.

B. Disruption vs. Detectability

Busy Loop (BL) is effective in reducing fuzzing throughput but is easily detectable via timing analysis. It may be suitable for quick deployment or testing environments, but lacks stealth.

Input-Dependent Delay (IDD) strikes a practical balance by injecting delays probabilistically. This introduces noise in execution behavior while maintaining high coverage—making it a viable option for stealthy defenses.

Dynamic Self-Modification (DSM) is the most potent. It halts fuzzing entirely by modifying control flow prior to parser invocation. However, this can interfere with debuggers, instrumentation tools, or cause program instability.

C. Deployment Considerations

- **Low-cost defense**: BL can be implemented rapidly with minimal changes.
- **Stealth defense**: IDD avoids consistent behavior that might be easily flagged.
- **Hard barrier defense**: DSM ensures maximum disruption but at potential operational cost.

D. Fuzzing Cost Imposition

All counter-fuzzing techniques aim to increase attacker cost:

- **BL** wastes CPU time per test.
- **IDD** introduces uncertainty and instability.
- **DSM** denies feedback altogether, likely causing early fuzzing failure.

E. Security Use Cases

These strategies are applicable in:

- Closed-source binaries defending against fuzzing-based reverse engineering.
- Embedded firmware with resource constraints.
- Systems requiring in-field obfuscation or protection from greybox testing.

VII. CONCLUSION AND FUTURE WORK

This work introduced and evaluated three novel counter-fuzzing strategies—Busy Loop, Input-Dependent Delay, and Dynamic Self-Modification—each targeting key mechanisms in coverage-guided fuzzers.

Our experiments show that:

- **BL** slows down fuzzing significantly with minimal engineering effort.
- **IDD** provides a stealthy disruption method, balancing exploration and delay.
- **DSM** is highly disruptive, capable of fully blocking fuzzing, albeit at a cost to instrumentation stability.

These techniques demonstrate that software can resist automated exploration by altering execution semantics or timing in targeted ways. This opens a new direction in software hardening that complements traditional access control or encryption.

Future Work

Future directions include:

- **Cross-fuzzer generalization**: Testing these strategies against concolic, grammar-based, and ML-driven fuzzers.
- **Dynamic counter-fuzzing**: Designing adaptive mechanisms that adjust defense levels at runtime.
- **Hybrid integration**: Combining fuzzing resistance with software diversity and runtime randomization.
- **Evaluation at scale**: Measuring effectiveness on large codebases and real-world applications.
- **Broader evaluation**: Applying these counter-fuzzing techniques to additional benchmarks such as `libpng`, `sqlite`, or `zlib`, and testing them against alternative fuzzers like `libFuzzer`, `honggfuzz`, and `Driller` to assess generalizability across fuzzing paradigms.

Table II
QUANTITATIVE SUMMARY OF FUZZING PERFORMANCE AND COUNTER-FUZZING EFFECTIVENESS

Strategy	Max Coverage	Mean Coverage	Total Execs	Crashes	Hangs	Avg Exec/sec	Disruption Ratio (DR)	Stealth Score (SS)
Baseline	16.57	16.55	3,297,644	0	0	646.98	0.000	0.25
BL	13.38	12.23	10,183	0	0	2.66	0.9959	0.02
IDD	17.07	17.01	2,539,676	0	0	533.24	0.176	0.09
DSM	0.03	0.03	1,298	0	0	11.10	0.9828	0.01

Ultimately, counter-fuzzing shifts the cost curve—making vulnerability discovery more expensive and inefficient for adversaries. As fuzzers evolve, so too must our defenses.

Ethical Considerations

While our techniques are designed to support defensive software hardening, they could potentially be misused by malicious actors to evade reverse engineering and fuzzing-based analysis. We urge responsible disclosure and controlled deployment of such methods, especially in security-sensitive environments. Transparency, reproducibility, and contextual safeguards are essential to prevent misuse.

Reproducibility

All implementation artifacts, evaluation scripts, and modified benchmark code will be released upon publication at [https://github.com/hyungminkimdev/counter-fuzzing]. If any restrictions prevent public release, we will provide access upon reasonable request.

REFERENCES

[1] M. Zalewski, “American Fuzzy Lop (AFL),” [Online]. Available: https://lcamtuf.coredump.cx/afl, 2015.

[2] K. Serebryany, “LibFuzzer: A library for coverage-guided fuzz testing,” in *Proc. LLVM Developers’ Meeting*, 2016.

[3] Google Open Source Blog, “OSS-Fuzz—Five Years Later: 10,500 Bugs and Counting,” 2022. [Online]. Available: https://opensource.googleblog.com/2022/05/oss-fuzz-five-years-later.html

[4] T. Petsios, A. D. Brown, A. A. Doupé, and G. Vigna, “SlowFuzz: Automatic domain-independent detection of algorithmic complexity vulnerabilities,” in *Proc. USENIX Security Symposium*, 2017.

[5] Y. Li, S. Yan, Z. Qian, and Z. Lin, “T-Fuzz: Fuzzing by program transformation,” in *Proc. IEEE Symp. Security and Privacy (S&P)*, 2019.

[6] Y. Qin, C. Song, M. Payer, and D. Song, “UnTracer: Differentially tracking uninstrumented binaries,” in *Proc. Network and Distributed System Security Symp. (NDSS)*, 2019.

[7] Google Research, “FuzzBench: Fuzzer benchmarking platform,” [Online]. Available: https://github.com/google/fuzzbench, 2020.