MES: Thwarting Fuzzing by Suppressing Memory Errors (Registered Report)

Fannv He[†‡*], Yuan Liu[§], Jice Wang[‡], Baiquan Wang[‡], Zezhong Ren[†], and Yuqing Zhang[†‡§✉]
[†]*National Computer Network Intrusion Protection Center, University of Chinese Academy of Sciences, China*
[‡]*School of Cyberspace Security, Hainan University, China*
[§]*School of Cyber Engineering, Xidian University, China*

***Abstract*—Fuzzing fundamentally relies on crash observability to guide its search. This paper breaks this premise by introducing *MES*, a novel anti-fuzzing system designed to make crashes unobservable. *MES* employs a compile-time address masking technique that instruments all memory accesses, ensuring they always refer to valid regions, thereby systematically suppressing memory-error crashes at their root. Our design stems from a validated foundational premise: invalid data accesses constitute the vast majority of crashes. Thus, a data-flow-centric suppression strategy offers the most effective defense. We evaluate *MES* through a three-pillar methodology: validating the premise via precise analysis of Binutils 2.13; assessing real-world efficacy against state-of-the-art fuzzers using the UNIFUZZ benchmark; and quantifying overhead/deployment scope with SPEC CPU 2017. *MES* is implemented as an LLVM compiler pass and a custom loader. Based on the experimental data obtained to date, *MES* demonstrates a strong capability to suppress memory-error crashes, with current results indicating a suppression rate exceeding 97% in our tests, which significantly impedes fuzzing progress. Preliminary performance measurements show that its overhead remains manageable within a well-defined operational envelope, supporting its promising potential as a practical defense in scenarios where crash suppression is critical. The full evaluation is ongoing to solidify these findings.**

I. Introduction

Fuzz testing has been widely used in automated bug hunting due to its proven efficiency and low entry cost. As researchers continue to develop sophisticated algorithms and techniques to further enhance fuzzing capability under different assumptions[1], [2], [3], [4], drag-and-drop fuzzing has already become increasingly feasible for binary-only fuzzing [5], [6], [7]. While these easy-to-use tools enable developers to identify code issues in-house, they can also be exploited by adversaries to harvest exploitable vulnerabilities in running code of deployed products, such as system components, client applications, and network services.

[*]Part of this work was done while Fannv He was at the University of Chinese Academy of Sciences, and part was done at Hainan University.
[✉]Corresponding author: Yuqing Zhang<zhangyq@nipc.org.cn>.

Fuzzing Workshop (FUZZING) 2026
27 February 2026, San Diego, CA, USA
ISBN 978-1-970672-04-6
https://dx.doi.org/10.14722/fuzzing.2026.23014
www.ndss-symposium.org

Anti-fuzzing efforts have been developed to protect production-ready programs from being effectively fuzzed. Prior anti-fuzzing techniques[8], [9], [10], [11] can be understood by the four fuzzing assumptions they target [12]: coverage feedback, application speed, solvable constraints and detectable crashes. While these approaches are effective against advanced fuzzers by depriving their intelligence (in test-case generation), with enough time and computing budget, a bug would eventually occur and be captured by the fuzzer. In other words, these efforts only slow down advanced fuzzers in discovering bugs. Unlike existing work, we propose undermining the single assumption that all fuzzers necessarily depend on: crash observability. Crash observability serves as the single assumption upon which all fuzzers rely for bug identification. Effective crash suppression can undermine the very foundation of fuzzing tools, rendering fuzzing tools impotent in their quest to uncover vulnerabilities. Although ANTIFUZZ [12] mentioned crash suppression, its core approach of hijacking signal processing functions can be easily bypassed with minimal engineering effort (e.g., by recognizing the invocation of signal registration functions and replacing them with no-operation instructions).

The solution proposed in this work, named *MES*[3], is a new anti-fuzzing technique that prevents crashes from occurring by suppressing memory errors, thwarting any fuzzer fundamentally. Our design is based on a simple observation: if we can ensure that memory references always point to valid memory regions, then the program is unlikely to crash and the fuzzer cannot observe the bug. To achieve this, we propose to utilize code instrumentation to normalize memory addresses. Specifically, we rearrange the data regions in aligned memory chunks by carefully crafting the memory layout of the target program. Then, we insert code instrumentation before memory access instructions to mask the target address so that the resulting address always falls within the aligned memory chunks, which are valid for access. *MES* is designed to suppress crashes caused by direct data accesses, leaving crashes caused by invalid control flow transfers (e.g., by jumping to an invalid target pointed to by corrupted control-data) unprotected. Although we can adopt a similar idea to mask control-flow transfer targets so that the program always jumps to valid code regions, the program behavior becomes non-deterministic

[3]*MES* is an acronym for Memory Error Suppressor.

and suspicious (e.g., a completely different execution trace) to adversaries. Even though invalid control flow transfers are not suppressed, we argue that it does not substantially degrade our anti-fuzzing capability. In fact, with the wide adoption of stack guards, many advanced control flow hijacking attacks begin with a normal data crash. Adversaries then leverage this crash to hijack the control data and further compromise the system. By effectively preventing the exploitation of common data crashes, we can significantly impede adversaries' ability to find exploitable entry points, thereby enhancing overall system security.

We have implemented a *MES* prototype for Linux. It is comprised of a customized compiler based on LLVM and a program loader called shelter. Our compiler includes an LLVM pass that inserts code instrumentation before memory access instructions. The output of the compilation is a special binary which we call binary core. The binary core cannot be loaded and executed by the default loader in the OS. Instead, we provide a shelter module that is responsible for creating the execution environment for the binary core. Specifically, it initializes the memory layout for the binary core and hooks on heap functions to make sure dynamic memory is arranged as desired (for optimized implementation).

Our implementation validates the foundational premise that invalid data accesses are the dominant cause of crashes. By suppressing these errors at compile time, *MES* effectively renders crashes unobservable to fuzzers. Preliminary evaluation confirms that it significantly impedes fuzzing progress while maintaining manageable overhead within a well-defined operational envelope, demonstrating its practical value for scenarios requiring reliable crash suppression.

The core contributions of this work are:

- **A Foundational Break**: We challenge and break the fundamental assumption in fuzzing, that of crash observability, by introducing *MES*, the first system designed to make crashes unobservable, thereby shifting the defense paradigm from detecting to preventing the very evidence fuzzers rely on.
- **A Practical, Root-cause Defense**: We implement this vision via a lightweight, compile-time address masking technique. By instrumenting all memory accesses to ensure they always target valid regions, *MES* systematically suppresses memory-error crashes at their source, offering a principled and effective protection mechanism.
- **A Validated and Scoped Solution:** Through a three-pillar evaluation (hypothesis validation, efficacy benchmarking and overhead profiling), we empirically validate our design premise and provide preliminary evidence that *MES* suppresses $> 97\%$ of crashes with manageable overhead within a well-defined operational envelope, offering practical guidance for its deployment.

II. Background

A. Software Fault Isolation

Our protection mechanism employs address masking to enforce address validity. This technique is an instance of Software Fault Isolation (SFI) [13], a well-established approach in software security. In fact, our design and implementation are largely inspired by Google's SFI implementation, namely Native Client (NaCl) [14]. Therefore we briefly introduce the necessary technical background on the SFI technique, and NaCl in particular. SFI builds an isolated sandbox for untrusted code so that an exploitation cannot evade and infect the host environment. Since it doesn't rely on any hardware feature, it has received wide adoption in many different architectures [15], [16], [17], [14], [18], [19], [20], [21], [22], [23]. SFI has been deployed in many real systems that isolate kernel modules [15], [24], [25], native code in JVM [26], [27], and browser plugins as done in Native Client [14], [20]. The main idea of SFI is to instrument the untrusted program so that it can only access data and run code within a specific region, called program fault domain. To achieve efficient implementation, SFI arranges the code region and data region in two continuous memory chunks. A data access has to fall into the specified data region while a control flow transfer has to go to the specified code region. To interact with external code, the untrusted program must explicitly call a well-protected interface, where strict scrutiny is conducted. Any attempts to access data or execute code outside of these regions will result in a fault and termination of the program.

There are two ways to enforce SFI, dynamic binary translation [28] and inlined reference monitor [14]. In the former, an interpreter dynamically translates the target code and adds security checking on the interpreted instructions at run-time. In the latter, the program is statically rewritten so that the inspection logic is inserted before the dangerous instructions, such as memory access instructions and control-flow transfer instructions. Compared to dynamic methods, SFI based on inlined reference monitor achieves better performance, especially when the source code is available. With source code, static analysis techniques can be leveraged to optimize the resulting binary. For example, when an instruction is deemed not to escape the sandbox, the reference monitor can be omitted. Native Client (NaCl), Google's efficient SFI to isolate browser plugins, is based on inlined reference monitor. In NaCl, a reference monitor is implemented by masking target addresses so that the results always fall into valid regions. To add address masking, NaCl only inserts one additional instruction to regulate a memory access or control-flow transfer. Because of this, in this paper, we base our implementation on NaCl.

B. Rationale for Only Suppressing Data Errors

Crashes can occur due to various reasons, such as invalid memory accesses, invalid control flow transfers, divide by zero, or illegal instructions. However, memory-related errors are the most common cause, accounting for the majority of crashes [29]. For instance, memory-related errors are more likely to be exploited to overwrite critical data, bypass authorization, or manipulate control-data to hijack control-flow. In fuzzing, a memory bug can be revealed as a data access error or a control flow transfer error. The latter occurs when control-data (such as a return address on the stack) is corrupted. While

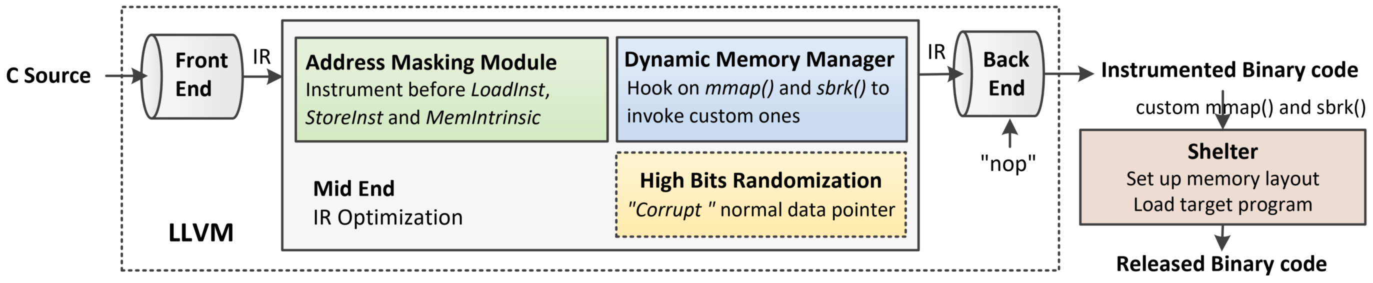

Fig. 1. *MES* Design Overview

the former cannot be directly used to hijack the control flow, they are frequently used in real-world attacks to indirectly control the program counter (PC). Additionally, they can be leveraged in non-control data attacks [30]. Therefore, we hypothesize that memory bugs caused by invalid data accesses are more common than those caused by invalid control flow transfers. In Section IV-A, we validate this hypothesis via real experiments. By suppressing data access errors, *MES* could effectively prevent many bugs from being revealed, including those that could later be exploited in a control flow hijacking attack. Direct suppression of invalid control flow transfers is also feasible. For instance, by aligning instruction boundaries and masking the program counter, all branches can be forced to a sanctioned address range within properly demarcated code [20]. However, the code's execution becomes unpredictable in the presence of corruption, making it easy for a skilled hacker to identify anomalies by tracking the execution trace. Moreover, while this approach may help suppress a few more crashes, it introduces much higher overhead in terms of memory consumption and performance. Hence, we focus solely on suppressing invalid data accesses in this work.

C. Threat Model

We assume an adversary who possesses the protected binary of the target program but has no access to its source code or any unprotected version from the development phase. The adversary is equipped with state-of-the-art fuzzing tools and is capable of employing both static and dynamic analysis to enhance the fuzzing process. His objective is to automatically uncover vulnerabilities in the binary. This threat model realistically reflects high-stakes scenarios where software is exposed to motivated attackers, such as in the case of industrial control systems or proprietary hardware-embedded programs. These systems often contain critical intellectual property or credentials, making them attractive targets and justifying the need for robust anti-fuzzing protection.

We further assume that the target binary is not instrumented with heavyweight memory safety tools such as AddressSanitizer[31]. While recent efforts like SanRazor[32] have made progress in reducing the overhead of such checkers, their runtime cost remains prohibitive for general deployment in production environments, thus leaving a practical window for deployable anti-fuzzing defenses.

III. Design and Implementation

Our goal is to design a protection mechanism that prevents crashes caused by memory errors during fuzzing. To this end, we introduce *MES*, which instruments the program to block unauthorized memory accesses.

A. Overview

MES comprises four major components: an address masking module, a dynamic memory manager, a high bits randomization module, and a shelter module. The overview of *MES* is shown in Fig. 1. First, an LLVM pass is used to insert an address masking operation before each memory access instruction so that the target address always points to a valid one (section III-B). To make the implementation more efficient, the valid memory region should be continuous whereas the default system library and OS make the data scattered across the virtual memory space. We address this problem with a custom memory manager to make sure all the dynamically allocated memory chunks, including those used by third-party libraries and heaps, fall within a continuous range which we call the *MES* Region (Section III-C). This module is provided as a library which will be linked with instrumented code, producing a protected binary, which we call *Binary core*. To defeat sophisticated attackers, we also randomize higher bits of the data pointers at their initialization points. If the address masking module is bypassed, the binary is self-destructed and cannot function correctly. This is implemented by a separate script that scans for pointer definitions (with the help of a symbol table) and makes the modifications directly in the binary (Section III-D). To accommodate the binary core on Linux, we introduce *Shelter*, an individual module that initializes the execution environment for the binary core (Section III-E). It allocates the *MES* Region, loads the binary core into the *MES* Region and finally transfers the control flow to the binary core.

B. Address Masking Module

There are two design choices available for implementing the memory error suppressor. The first approach involves using a memory error checker to compare the target address with the intended pointer bounds and deny any access that falls outside the legitimate range. If an out-of-bound access is detected, we have the option to update it by returning a random address within the bounds, as shown in function `MES_1()` in Listing 1. However, in our threat model, the binary core is considered public, which makes it easier for attackers to disarm the instrumentation(e.g., by replacing the inserted memory checkering with `nop` instructions). Additionally, memory error checkers typically impose higher performance and memory overhead [31].

To address these issues, we follow an alternative solution – using SFI to redirect illegal memory accesses. Technically, before each memory access instruction, a mask is unconditionally applied to the target address so that the result always points to a predefined *MES* region. This approach is demonstrated in function `MES_2()` in Listing 1. In the listing, `MB` (*MES* Begin) refers to the start address of the *MES* region and `ME` (*MES* End) refers to the end address of the *MES* region. 2^{MLEN} is the size of the *MES* region. As can be seen, after `MES_2()`, the target pointer would always fall in the *MES* region and retain the last `MLEN` bits.

We use an example to better explain the idea. Assuming that `ptr` was manipulated to `0x2345ac00` during fuzzing, the instruction `*ptr = val` (line 7) would result in an invalid memory access and crash the execution. After instrumentation, the resulting address would become `0x1234ac00`, which is a valid address.

```
 1  // Target code
 2  ...
 3  int *ptr, val;
 4  ...
 5
 6  + ptr = MES_1|2(ptr);//instrumentation
 7  *ptr = val;
 8  ---------------------------------------------
 9  // Instrumentation implementation
10  #define MB 0x12340000
11  #define ME 0x1234FFFF
12  #define MLEN 16
13  #define MES_MASK ((1 << MLEN) - 1)
14
15  // Option 1: Use memory checker
16  template <typename T>
17  T* MES_1(T *ptr) {
18    if (ptr < MB || ptr > ME)
19      return MB + rand(MES_MASK);
20    return ptr;
21  }
22
23  // Option 2: use address masking
24  template <typename T>
25  T* MES_2(T *ptr) {
26    T* MES_ID = ((MB >> MLEN) << MLEN);
27    return ((ptr & MES_MASK) | MES_ID);
28  }
```

Listing 1. Instrumenting memory accesses

The aforementioned idea can be easily implemented via an LLVM pass over the IR code. Specifically, we scan for all the memory accessing instructions (including StoreInst, LoadInst and MemIntrinsic) and correspondingly insert the memory masking logic. Note that when applying the mask, we do not consider the length of the buffer to be accessed. For example, *memset(ptr, 0, size)* might cause a crash when 1) *ptr* is invalid, and 2) the size exceeds the boundary (i.e., `ptr + size` is invalid). Our prototype implementation only applies the mask to the base address and pads a 1KB buffer after the *MES* region, assuming that the overflow cannot exceed a 1KB limit. This implementation choice significantly improves the instrumentation efficiency while allowing crashes to occur in very rare cases.

C. Dynamic Memory Manager

As mentioned before, *MES* redirects all memory accesses to a continuous *MES* region, including dynamically allocated memory buffers. Dynamic memory is requested via the *mmap* and *sbrk* function calls to the kernel. Therefore, we implemented our custom implementation of *mmap* and *sbrk* and wrapped them into *wrapper_mmap* and *wrapper_sbrk*. These custom implementations make sure that the returned pointers are always within the *MES* region. Finally, we locate all the callsites of *mmap* and *sbrk* and patch them to redirect the calls to our custom implementation. This module was implemented in the C programming language and linked to the target program using a clang wrapper called *mes-clang*.

D. High Bits Randomization

The memory masking module is effective in preventing memory errors. However, after the binary has been distributed to users, it can be bypassed by experienced reverse-engineers. For instance, an attacker can replace all the instrumentation with `nop` instructions. To prevent our protection from being bypassed, we designed a self-deconstruct mechanism to thwart advanced attackers. The idea is to intentionally "corrupt" a normal pointer so that if the instrumentation is removed, the program will crash. This is achieved by populating the higher bits of normal pointers. To make it indistinguishable from a genuinely corrupted pointer, we randomize the values.

We apply high bits randomization to two types of pointers. First, some data pointers are hard-coded in the binary. We scan for the relocation table to find their locations in the binary. Note the location of these data pointers can only be determined after linking and therefore must appear in the relocation table. Second, for dynamically allocated memory, we directly randomize the higher bits of the returned pointers.

We use an example to better explain the idea. Suppose the *MES* Region ranges between *0x12340000* to *0x1235000*. If the dynamic memory manager returns `0x1234f000` for a memory request, after randomization, the return address might be *0xadecf000*, which seems to be an illegal address. However, after address masking, the program would use `0x1234f000` instead and the program would run properly. Now assume that the attacker attempts to null-fill the address masking instrumentation with *nop* instructions. In this case, even though a memory error can potentially lead to a program crash, the program will not function properly since a "normal" memory access would trigger a segment fault.

E. Shelter

The shelter is the loader for the binary core, and itself is loaded by the system-provided loader. Once it has prepared

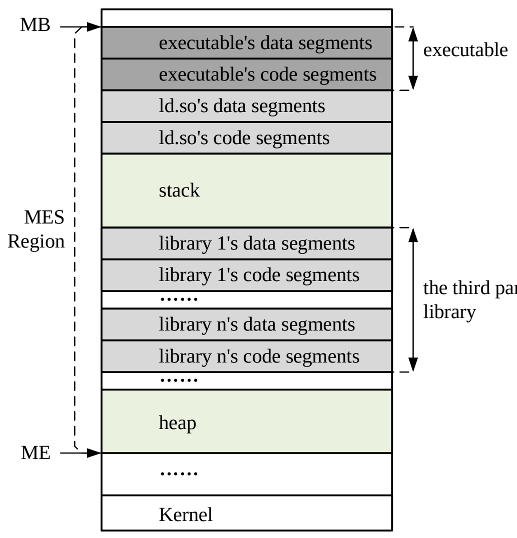
MB
executable's data segments
executable's code segments
executable
ld.so's data segments
ld.so's code segments
stack
MES Region
library 1's data segments
library 1's code segments
......
library n's data segments
library n's code segments
the third party library
......
heap
ME
......
Kernel

Fig. 2. Memory layout inside the *MES* region

the execution environment for the binary core, it hands over control to the latter, which runs the main program function.

Internally, it first allocates a large contiguous memory space called the *MES* Region. Starting from the start address, the binary core and `ld` are loaded, followed by a reserved stack, memory for third-party libraries and heap, etc. The memory layout inside the *MES* region is shown in Fig.2. Then, the shelter transfers control to the binary core, which uses `ld` to load required libraries. Note that we replace the dynamic memory allocator in `ld` with our own, as discussed in Section III-C. This ensures that the third-party libraries are loaded within the *MES* Region.

IV. Evaluation Plan

This section describes the experimental plan for evaluating our method. We aim to answer the following four research questions through the evaluation:

RQ1. Is our foundational premise regarding invalid data accesses constituting the vast majority of practical crashes empirically valid?

RQ2. How effective is *MES* in suppressing crashes against state-of-the-art fuzzers?

RQ3. What is the overhead and effective deployment scope of *MES*?

RQ4. what are the key advantages of *MES* over the current state-of-the-art anti-fuzzing tool?

A. Benchmarks to Use

To address our four research questions, we employ a trio of benchmark sets, each carefully selected to provide the most direct and relevant evidence for a specific subset of RQs.

- *Binutils 2.13 Program Set (For Hypothesis Validation: RQ1).* To directly examine the soundness of our core hypothesis that crashes are predominantly data-flow driven, we require an analyzable environment. We select 4 classic utilities (nm, objdump, objcopy, readelf) from Binutils 2.13. Its simpler, legacy codebase, lacking modern hardening, is ideal for fine-grained tracing of input propagation and unambiguous root-cause classification, thus specifically serving RQ1.
- *UNIFUZZ Benchmark Suite (For Effectiveness Comparison: RQ2).* To evaluate the practical crash-suppression effectiveness and identify key advantages over state-of-the-art fuzzers, we need a diverse set of modern, real-world programs. We adopt the UNIFUZZ benchmark, comprising 20 programs across domains like file parsing and multimedia processing. Its scale and complexity provide the necessary context to answer RQ2 (effectiveness) and RQ3 (advantages) under realistic conditions.
- *SPEC CPU 2017 Benchmark Suite (For Overhead and Deployment Scope: RQ3).* Quantifying performance overhead and determining the practical deployment boundaries require measuring resource usage on standardized, computationally intensive workloads. We utilize SPEC CPU 2017, a industry-standard performance benchmark, to obtain precise measurements of CPU and memory overhead under sustained execution. This data is essential to answer RQ4 regarding performance cost and viable scope of deployment.

TABLE I
Crash Causes: Invalid Data Access vs. Invalid Control Flow Transfer

Target	Total Crashes	Inv. Control Flow Transfer		Inv. Data Access		Others	
		Num.	Rates	Num.	Rates	Num.	Rate
nm	12,259	1,161	9.47%	11,097	90.52%	1	0.01%
objcopy	15,247	36	0.24%	15,211	99.76%	0	0.00%
objdump	10,692	1,505	14.08%	9,186	85.91%	1	0.01%
readelf	44,627	0	0.00%	44,622	99.99%	5	0.01%
Total	82,825	2,702	**3.26%**	80,116	**96.73%**	7	0.01%

This three-pillar benchmark strategy ensures that each research question is addressed with the most appropriate experimental subject, allowing for targeted, unambiguous, and comprehensive evaluation.

B. RQ1. Is our foundational premise regarding invalid data accesses constituting the vast majority of practical crashes empirically valid?

To empirically validate our foundational premise that invalid data accesses constitute the vast majority of software crashes, we conducted a controlled experiment. We used HONGGFUZZ [33] to fuzz four standard applications (nm, objdump, objcopy, readelf) from *binutils* 2.13 for 72 hours, precisely tracking and categorizing all unique crashes.

The results, detailed in Table I, provide definitive validation: 96.73% of all unique crashes stem from invalid data accesses, while invalid control-flow transfers account for a negligible 3.27%. This overwhelming distribution confirms that a crash suppression strategy focused on data-access errors directly addresses the predominant source of failures. It thereby establishes a highly efficient and empirically grounded foundation for our system design, promising maximum impact with manageable overhead and complexity.

TABLE II
COMPARISON OF CRASHES BEFORE AND AFTER *MES* PROTECTION

Target Binary	AFL++ (QEMU mode)				QSYM (with AFL++ QEMU)				HONGGFUZZ			
	Total Inputs	Before	After MES	Protection Ratio (%)	Total Inputs	Before	After MES	Protection Ratio (%)	Execution Count	Before	After MES	Protection Ratio (%)
tiffsplit	11000	321	35	89.09	7903	740	31	95.81	57133572	31	0	100.00
cflow	2509	277	0	100.00	13600	348	0	100.00	34046103	2241	2	99.91
jhead	5642	87	0	100.00	6072	213	0	100.00	73884766	8	0	100.00
Average	-	685	35	94.89	-	1301	31	97.62	-	2280	2	99.91

C. RQ2: How effective is MES in suppressing crashes against state-of-the-art fuzzers?

To quantitatively assess *MES*'s crash suppression capability (RQ2), we design an empirical evaluation based on the UNIFUZZ benchmark suite. This evaluation is structured as follows:

- *Adversarial Fuzzers*: To challenge *MES*, we employ three state-of-the-art fuzzers configured for maximum coverage and depth: AFL++ (in QEMU mode), QSYM (configured to work with AFL++ QEMU), HONGGFUZZ (in QEMU mode).
- *Experimental Procedure*: For each target program, we will conduct a 72-hour fuzzing campaign in parallel on two versions: the original program and its *MES*-protected version. Each fuzzer-program pair will be executed in a fresh, isolated environment to ensure fairness and reproducibility.
- *Metric*: The primary metric is the Crash Suppression Rate. For each run, we will collect and triage all unique crashes. The effectiveness of *MES* will be calculated by comparing the number of unique crashes found in the protected version against the baseline (original version).
- *Analysis*: The compiled data, including the per-program and aggregate crash suppression rates, will be presented and analyzed to determine *MES*'s overall efficacy and its robustness against diverse, advanced fuzzing strategies.

The partial data available thus far (summarized in Table II) indicate a promising average crash suppression rate of 97.47% for *MES*. We will report the complete and finalized dataset upon conclusion of the experiment.

D. RQ3. What is the overhead and effective deployment scope of MES?

To quantitatively answer RQ3 regarding the overhead and effective deployment scope of *MES*, we design a two-phase evaluation. This methodology moves from isolated component costing to a data-informed definition of practical applicability, ensuring that deployment guidance is rooted in empirical performance profiles.

Phase 1: Component-Wise Overhead Profiling via SPEC CPU 2017.

We will measure the performance overhead using the industry-standard SPEC CPU 2017 benchmark suite under the refrate configuration. To precisely attribute costs, a three-stage instrumentation approach will be employed:

- *Core Protection*: Base Address Masking Instrumentation. Measure the theoretical performance impact of the core address randomization mechanism.
- *+ Heap Manager*: Core Protection + Custom Heap Manager. Quantify the performance change introduced by dynamic memory management optimization.
- *Full System*: Core Protection + Heap Manager + Shelter Loader. Evaluate the operational efficiency of the complete solution in a realistic deployment environment.

For each stage, selected C-language benchmarks from the SPECrate Integer and Floating-Point suites will be compiled, run, and compared against a native, unprotected baseline. Results will be reported as percentage slowdown per benchmark and as geometric means per category, providing a granular cost breakdown.

Phase 2: Defining Deployment Scope from Empirical Profiles. The overhead data from Phase 1, particularly its variation across workload types (e.g., compute- vs. memory-intensive), serves as the primary input for scoping. We will systematize this mapping along three dimensions to translate performance observations into deployment guidelines:

- *Compatibility Constraints*: Documenting hard requirements (toolchain, architecture) and limitations.
- *Performance-Deployment Correlation*: Explicitly linking high-overhead profiles (e.g., memory-intensive loops) to specific application contexts where *MES* may be less suitable, and low-overhead profiles (e.g., logic/network-bound programs) to its optimal use cases.

Through this two-phase process, we will not only quantify the performance cost composition of *MES* but, more critically, derive its effective deployment scope from that empirical cost analysis. This provides practitioners with clear, evidence-based guidance on where *MES* offers a favorable security-performance trade-off.

Table III summarizes the partial experimental data obtained thus far. We observed the impact on CPU-intensive applications was lower(e.g., 519.lbm_r with -0.16%). 519.lbm_r is a fluid dynamics calculation method, a computationally intensive application, and involves few memory operations. *MES* has little impact on it, only -0.16%. Other benchmarks (e.g., 557.xz r with 26.07%) involve more memory operations, and instrumenting will cause a lot of performance overhead. Further analysis reveals a positive correlation between the number of memory access instructions in a program and the performance overhead introduced by *MES*. Since we insert address masking

TABLE III
SPEC CPU2017 BENCHMARK. THE EXPERIMENT WAS DIVIDED INTO THREE GROUPS: CLANG, HEAP, *MES*. CLANG: NO PROTECTION MECHANISM. HEAP: ONLY USE SHELTER AND CUSTOM HEAP MANAGEMENT. MES: ADD ADDRESSS MASK INSTRUMENTATION BASED ON HEAP.

Benchmarks	Type	Clang(s)	Heap		*MES*	
			Time(s)	Overhead(%)	Time(s)	Overhead(%)
557.xz_r	SPECrate Integer	462.21	462.88	0.14%	582.73	26.07%
519.lbm_r	SPECrate Floating Point	321.70	317.63	-1.27%	321.20	-0.16%

TABLE IV
CRASH SUPPRESSION METHODS: ANTIFUZZ VS. MES

Property	ANTIFUZZ	MES
Anti-Fuzzing	✓	✓
Core Strategy	Reactive (Hides Crashes)	Proactive (Prevents Crashes)
Anti-RE	×	✓
Self-Contained	×	✓
Bypass Risk	High	Low

operations before every memory load instruction, the processor must perform additional address calculations before accessing memory. In our current implementation, instrumentation is applied to all memory access instructions, which inevitably introduces redundancy in certain scenarios. For instance, when a program iterates through an array, the number of masking operations becomes proportional to the array length, leading to a substantial increase in runtime overhead. However, for applications with infrequent memory accesses, *MES* achieves effective crash suppression with minimal performance impact. Take network protocol parsers as an example: in such control logic-intensive programs, the majority of execution time is spent on logical decisions and state processing, while actual memory write operations remain relatively limited. In these scenarios, *MES* introduces only a negligible number of masking instructions, resulting in almost imperceptible performance degradation while still effectively preventing the majority of memory error-induced crashes. This demonstrates the remarkable practical value and deployment advantages of *MES* in suitable use cases. The experiment remains ongoing, and the full dataset will be reported upon its completion.

E. RQ4. what are the key advantages of MES over the current state-of-the-art anti-fuzzing tool?

Research dedicated specifically to crash suppression remains scarce. To the best of our knowledge, ANTIFUZZ [34] and our proposed *MES* are the only two dedicated works in this domain, yet they are founded on diametrically different designs. To clearly delineate our contribution and answer RQ4, we compare these two approaches. As summarized in Table IV, while both systems achieve the core objective of anti-fuzzing, *MES* exhibits fundamental advantages across critical dimensions due to its proactive-preventive paradigm.

Specifically, ANTIFUZZ adopts a reactive strategy of crash masking via signal handler hijacking. This approach is inherently fragile: it introduces a high bypass risk (e.g., by avoiding signal invocation), lacks integrated anti-RE protections, and is not self-contained, relying on external toolchain dependencies. In contrast, *MES* is built on a proactive, preventive paradigm. By randomizing the memory layout via address masking, it fundamentally raises the barrier to triggering crashes, leading to low bypass risk. This design inherently impedes reverse engineering (Anti-RE) and operates as a fully self-contained solution. These combined advantages make *MES* more robust, deployable, and secure in practice, justifying its architectural departure from the prior state-of-the-art.

V. DISCUSSION

a) Control flow protection: *MES* is designed to suppress crashes caused by invalid direct data accesses, but does not handle those resulting from illegitimate control-flow transfers, such as execution jumps to an invalid address referenced by corrupted control data. While a similar masking approach could be applied to constrain control-flow targets to valid code regions, doing so would lead to non-deterministic and semantically inconsistent program behavior. Although such control-flow violations remain unmitigated, we argue that this does not materially compromise *MES* 's anti-fuzzing capabilities, a conclusion supported by the experimental results in Section IV-B. Existing control-flow protection mechanisms, such as stack canaries and Control-Flow Integrity (CFI), have significantly improved the resilience of software against exploitation. During fuzzing, crashes caused by control-flow hijacking are often non-exploitable, as these mechanisms effectively contain such errors. Thus, enabling standard control-flow protection in compiled binaries is generally sufficient to mitigate such issues. However, mature and reliable data-flow protection mechanisms remain notably lacking in both research and practice. *MES* is therefore designed specifically to address this gap by focusing on data-flow protection, thereby impeding state-of-the-art fuzzers from identifying crashes stemming from data-flow errors. By deploying *MES*, the majority of data-flow related crashes are suppressed, limiting fuzzer observations primarily to non-exploitable control-flow violations.

b) Third party libraries: Executable files often rely on third-party libraries. Without access to source code, the protection offered by *MES* cannot be extended to these external libraries. As a result, the program remains vulnerable to crashes when executing instructions within such libraries. For instance, certain functions take pointers as parameters, which may reference data outside the protected memory regions, such as data located in third-party libraries. Since these libraries lie beyond the scope of *MES* 's protection, we cannot guarantee the safety of memory accesses performed by them. This limitation is inherent to instrumentation-based approaches.

Furthermore, *MES* does not protect parameters of pointer type. Pointers may refer not only to data regions but also to function addresses, making it difficult to distinguish between data flow and control flow. As a result, pointer-type parameters can still lead to crashes. During experiments, we also observed crashes caused by overwriting length parameters of certain string and memory functions, leading to accesses beyond valid memory regions. We attempted to mitigate these issues by hooking such functions but eventually refrained from doing so to avoid overfitting the solution to specific cases.

c) Bypassing protection: A potential concern is that our *MES* protection could be circumvented by skilled adversaries using well-known techniques such as static binary rewriting or dynamic library injection. However, both approaches depend on accurately recovering program semantics, which presents a significant challenge when only the binary is available. While identifying heap-based errors is relatively straightforward, precisely locating the bounds of stack-allocated arrays and structures remains considerably more challenging. In fact, accurately inferring such stack layout information from binaries is extremely difficult and, to the best of our knowledge, no practical solution exists today. Related binary instrumentation systems, such as RetroWrite [35], which supports AFL and Address Sanitizer (ASan), also operate under specific assumptions and do not fully resolve this issue.

VI. Related Work

a) Memory error checker: Memory errors such as out of bounds array accesses and invalid pointer accesses are a common source of program failures. Safe languages such as ML and Java use dynamic checks to eliminate such errors: for example, if the program attempts to access an out of bounds array element, the implementation intercepts the attempt and throws an exception. The rationale is that an invalid memory access indicates an unanticipated programming error and it is unsafe to continue the execution without first taking some action to recover from the error. Although *MES* does not do memory error checking, it can ensure that most memory errors will not make the program crash and avoid fuzzer finding more bugs. Like *MES*, many works relying on memory layout information implements a safe-C compiler, which dynamically checks that intercept out of bounds array accesses and accesses via invalid pointers [36], [37]. Rinard et al. [38] present failure-oblivious computing, which make servers invulnerable to known security attacks that exploit memory errors, and enable the servers to continue to operate successfully to service legitimate requests and satisfy the needs of their users even after attacks trigger their memory errors. Despite the different goals, there are many similarities between failure-oblivious computing and *MES*.

b) Anti-Fuzzing: Anti-fuzzing techniques thwart the misuse of fuzzing through binary hardening [12], [8], execution environment maneuver [12], [39] or both. Miller believes that anti-fuzzing should be divided into two stages. Firstly, the application of anti-fuzzing technique detects fuzzing, and secondly, defensive measures are taken before fuzzing starts [40]. Hu et al. proposed the Chaff Bugs technique, which inserts many unusable bugs into the program, thus wasting the attacker's time and energy [41]. Edholm introduced four aspects of anti-fuzzing technique: execution speed, masking crashes, detection of the fuzzing framework, and made a simple evaluation to verify the feasibility of the scheme, but there is no complete system implementation [39]. Jung et al. proposed three technologies: SpeedBump, BranchTrap, and AntiHybrid. They attacked the three characteristics of fuzzing execution speed, coverage information feedback, and symbolic execution. Users can freely combine these three technologies to achieve an excellent anti-fuzzing effect [8]. Güler et al. put forward four underlying assumptions that fuzzing obeys: coverage feedback, detectable crashes, application speed, and solvable constraints; four countermeasures are proposed for these four underlying assumptions: attacking coverage-guidance, preventing crash detection, delaying execution, overloading symbolic execution engines [12]. *MES* uses code instrumentation technology and replace the default loader with Shelter. Different from the existing work, *MES* does not dynamically identify fuzziers and prevent them. It doesn't inject some fake bugs to distract attackers and waste their time and energy. *MES* does not interfere with feedback mechanism, reduce execution speed, attack symbol execution and other auxiliary measures. *MES* only focuses on the core assumption of fuzzing to find vulnerabilities, and tries to fight against fuzzing in the most direct way. *MES* compulsorily accesses legal memory area through address mask to reduce the possibility of program crash due to memory error, reduce the probability of fuzzer detect crash, and effectively prevent fuzzing.

VII. Conclusion

In this paper, we introduced *MES*, a novel anti-fuzzing system that fundamentally disrupts fuzzing by breaking its core reliance on crash observability. Unlike prior obfuscation or slowdown techniques, *MES* proactively suppresses memory-error crashes at the source through compile-time address masking, making bugs effectively unobservable to fuzzers. This work rests on the validated premise that data access errors dominate real-world crashes, and is implemented through practical, LLVM-based instrumentation and a dedicated loader. Our preliminary evaluation demonstrates that *MES* effectively suppresses the vast majority of crashes (>97%) with manageable overhead within a well-defined operational envelope, establishing a new point in the design space of reliable, mechanism-based anti-fuzzing defenses.

Acknowledgment

We would like to thank Han Zheng for his helpful input and suggestions on the experiment. This work was supported by the National Key Research and Development Program of China (2023YFB3106400), the National Natural Science Foundation of China (U2336203, 62562025), the Beijing Natural Science Foundation (L242015, 4242031), and the Innovation Fund of Xidian University (GNYZ2024QC009).

References

[1] C. Wang, W. Meng, C. Luo, and P. Li, "Predator: Directed web application fuzzing for efficient vulnerability validation," in *2025 IEEE Symposium on Security and Privacy (SP)*, 2025, pp. 886–902.

[2] M. Ammann, L. Hirschi, and S. Kremer, "Dy fuzzing: formal dolev-yao models meet cryptographic protocol fuzz testing," in *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2024, pp. 1481–1499.

[3] H. Huang, P. Yao, H.-C. Chiu, Y. Guo, and C. Zhang, "Titan : Efficient multi-target directed greybox fuzzing," in *2024 IEEE Symposium on Security and Privacy (SP)*, 2024, pp. 1849–1864.

[4] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation." New York, NY, USA: Association for Computing Machinery, 1993. [Online]. Available: https://doi.org/10.1145/168619.168635

[5] D. M. Vincent Ulitzsch, Bhargava Shastry, "Follow the white rabbit simplifying fuzz testing using fuzzexmachina," https://i.blackhat.com/us-18/Thu-August-9/us-18-Ulitzsch-Follow-The-White-Rabbit-Simplifying-Fuzz-Testing-Using-FuzzExMachina.pdf, 2018, accessed: 2020-08-10.

[6] Google, "Fuzzing for security," https://blog.chromium.org/2012/04/fuzzing-for-security.html, 2012, accessed: 2020-08-10.

[7] M. Aizatsky, K. Serebryany, O. Chang, A. Arya, and M. Whittaker, "Announcing oss-fuzz: Continuous fuzzing for open source software," *Google Testing Blog*, 2016.

[8] J. Jung, H. Hu, D. Solodukhin, D. Pagan, K. H. Lee, and T. Kim, "Fuzzification: Anti-fuzzing techniques," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1913–1930.

[9] J. Zhang, Z. Li, Y. Liu, Z. Sun, and Z. Wang, "Safte: A self-injection based anti-fuzzing technique," *Computers and Electrical Engineering*, vol. 111, p. 108980, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045790623004044

[10] H. Y. Kim and D. H. Lee, "Catchfuzz: Reliable active anti-fuzzing techniques against coverage-guided fuzzer," *Computers & Security*, vol. 143, p. 103904, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404824002062

[11] P. Liu, Y. Zheng, C. Sun, H. Li, Z. Li, and L. Sun, "Battling against protocol fuzzing: Protecting networked embedded devices from dynamic fuzzers," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 4, Apr. 2024. [Online]. Available: https://doi.org/10.1145/3641847

[12] E. Güler, C. Aschermann, A. Abbasi, and T. Holz, "Antifuzz: Impeding fuzzing audits of binary executables," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 1931–1947.

[13] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proceedings of the fourteenth ACM symposium on Operating systems principles*, 1993, pp. 203–216.

[14] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 2009, pp. 79–93.

[15] C. Small, "A tool for constructing safe extensible c++ systems," in *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*, 1997.

[16] S. McCamant and G. Morrisett, "Evaluating sfi for a cisc architecture." in *USENIX Security Symposium*, 2006.

[17] B. Ford and R. Cox, "Vx32: Lightweight user-level sandboxing on the x86." in *USENIX Annual Technical Conference*. Boston, MA, 2008, pp. 293–306.

[18] B. Zeng, G. Tan, and G. Morrisett, "Combining control-flow integrity and static analysis for efficient and validated data sandboxing," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 29–40.

[19] B. Zeng, G. Tan, and Ú. Erlingsson, "Strato: A retargetable framework for low-level inlined-reference monitors," in *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, 2013, pp. 369–382.

[20] D. Sehr, R. Muth, C. L. Biffle, V. Khimenko, E. Pasko, B. Yee, K. Schimpf, and B. Chen, "Adapting software fault isolation to contemporary cpu architectures," 2010.

[21] L. Deng, Q. Zeng, and Y. Liu, "Isboxing: An instruction substitution based data sandboxing for x86 untrusted libraries," in *IFIP International Information Security and Privacy Conference*. Springer, 2015, pp. 386–400.

[22] L. Zhao, G. Li, B. De Sutter, and J. Regehr, "Armor: fully verified software fault isolation," in *Proceedings of the ninth ACM international conference on Embedded software*, 2011, pp. 289–298.

[23] Y. Zhou, X. Wang, Y. Chen, and Z. Wang, "Armlock: Hardware-based fault isolation for arm," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 558–569.

[24] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "Xfi: Software guards for system address spaces," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 75–88.

[25] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Software fault isolation with api integrity and multi-principal modules," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, pp. 115–128.

[26] J. Siefers, G. Tan, and G. Morrisett, "Robusta: Taming the native beast of the jvm," in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010, pp. 201–211.

[27] M. Sun and G. Tan, "Jvm-portable sandboxing of java's native libraries," in *European Symposium on Research in Computer Security*. Springer, 2012, pp. 842–858.

[28] M. Payer and T. R. Gross, "Fine-grained user-space security through virtualization," *ACM SIGPLAN Notices*, vol. 46, no. 7, pp. 157–168, 2011.

[29] M. Miller, "Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape," *BlueHat IL*, pp. 11–19, 2019.

[30] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 969–986.

[31] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker," in *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, 2012, pp. 309–318.

[32] J. Zhang, S. Wang, M. Rigger, P. He, and Z. Su, "SANRAZOR: Reducing redundant sanitizer checks in C/C++ programs," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, Jul. 2021, pp. 479–494.

[33] R. Swiecki, "Honggfuzz," *Available online at: http://code. google. com/p/honggfuzz*, 2016.

[34] Y. Zhou, H. M. G. Wassel, S. Liu, J. Gao, J. Mickens, M. Yu, C. Kennelly, P. Turner, D. E. Culler, H. M. Levy, and A. Vahdat, "Carbink: Fault-Tolerant far memory," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 55–71.

[35] S. Dinesh, "Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization," Ph.D. dissertation, Purdue University Graduate School, 2019.

[36] S. H. Yong and S. Horwitz, "Protecting c programs from attacks via invalid pointer dereferences," in *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, 2003, pp. 307–316.

[37] O. Ruwase and M. S. Lam, "A practical dynamic buffer overflow detector." in *NDSS*, vol. 2004, 2004, pp. 159–169.

[38] M. C. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, "Enhancing server availability and security through failure-oblivious computing." in *Osdi*, vol. 4, 2004, pp. 21–21.

[39] E. Edholm and D. Göransson, "Escaping the fuzz-evaluating fuzzing techniques and fooling them with anti-fuzzing," Master's thesis, 2016.

[40] C. Miller, "Anti-fuzzing," *Unpublished. Available: https://www. scribd. com/document/316851783/anti-fuzzing-pdf*, 2010.

[41] Z. Hu, Y. Hu, and B. Dolan-Gavitt, "Chaff bugs: Deterring attackers by making software buggier," *arXiv preprint arXiv:1808.00659*, 2018.