QMSan: Efficiently Detecting Uninitialized Memory Errors During Fuzzing

Matteo Marini*, Daniele Cono D'Elia*, Mathias Payer† and Leonardo Querzoni*
*Sapienza University of Rome
†EPFL
{m.marini, delia, querzoni}@diag.uniroma1.it, mathias.payer@nebelwelt.net

***Abstract*—Fuzzing evolved into the most popular technique to detect bugs in software. Its combination with sanitizers has shown tremendous efficacy in uncovering memory safety errors, such as buffer overflows, that haunt C and C++ programmers. However, an important class of such issues, the so-called *use-of-uninitialized-memory* (UUM) *errors*, struggles to gain similar benefits from fuzzing endeavors. The only fuzzer-compatible UUM sanitizer available to date, MSan, requires that all libraries are fully instrumented. Unlike address sanitization, for which partial instrumentation results in false negatives (missed detection of bugs), UUM sanitizers require complete instrumentation to avoid false positives, hampering testing at scale. Yet, full-stack compiler-based instrumentation can be a daunting prospect for compatibility and practicality. As a result, many programs are left untested for UUM bugs.**

In this paper, we propose an efficient multi-layer, opportunistic design that does not require (source-based) recompilation of all code without harming accuracy. The multiplicity of executions when fuzzing offers us the opportunity to learn what any encountered false positive looks like, and later ignore them when we meet them again with new test cases. Such an avenue is feasible only if one can resort to fast techniques to effectively discriminate candidate errors, or false negatives will then occur.

We show how to realize this design by using the dynamic binary translation of QEMU for compatibility and lightweight code analysis techniques to achieve scalability and accuracy. As a result, we obtain a fuzzer-friendly, performant sanitizer, QMSAN, that effectively tackles current practicality challenges of UUM error detection. On a collection of 10 open-source and 5 proprietary programs, QMSAN exposed 44 new UUM bugs. In our tests, QMSAN incurs slowdowns of 1.51x over QEMU and 1.55x over the compiler-based instrumentation of MSan, showing no false positives and false negatives. QMSAN is open-source.

I. Introduction

Programming languages like C and C++ remain prevalent despite being prone to memory errors. While safer alternatives exist, most software is implemented in unsafe languages due to performance requirements or the need to access low-level systems features. In C and C++, programmers are responsible for ensuring memory safety and must take care not to introduce buffer overflow, use-after-free, or double-free vulnerabilities along with proper initialization of all memory. As these bugs often have security implications, exposing and mitigating memory safety errors remains a crucial area of research.

Use-of-uninitialized-memory (UUM) errors occur when the outcome of a program computation depends on an indeterminate value. A program reads an indeterminate value from a storage location when no prior assignment initialized it fully, or at all, since its declaration [1]. Like other memory safety errors, UUM errors may be difficult to detect because they do not necessarily result in conspicuous behavior (e.g., program crashes) but may subtly and silently corrupt the program state, and come with an unpleasant property of unpredictability [2]. For example, an uninitialized stack variable may show left-over data from stale stack frames from prior function calls. Unfortunately, in the hands of a capable attacker, these errors may become exploitable, as witnessed for notable UUM bugs that enabled information disclosure, remote code execution, and guest-to-host privilege escalation [2], [3], [4], [5], [6].

Software sanitizers effectively detect memory safety errors. These tools safeguard one or more memory safety properties at run-time by instrumenting program code with tripwires that expose safety violations during execution. As dynamic analysis tools, sanitizers suffer from the coverage problem and require that the bug is reached and triggered through an actual execution to detect it. Therefore, they are very effective in combination with techniques like fuzzing, which broadly explores the program [7]. For UUM errors, currently, the only way to detect them when fuzzing is using Google's MSan [8]. Similarly to other popular sanitizers like ASan [9], MSan uses compiler instrumentation to insert run-time checks on data uses to check if their sources were properly initialized.

MSan faces a key practical limitation: all code under test must be compiled with MSan instrumentation. Looking at the C/C++ programs tested daily by the OSS-Fuzz initiative, we note that only 210 out of 528 (40.5%) show MSan support at the time of writing. According to the OSS-Fuzz bug tracker, though, MSan was helpful to expose over 289 UUM bugs in them in the past 18 months. We also recall a notable setback when OSS-Fuzz upgraded its backend in 2021 to Ubuntu 20.04 [10], which led its maintainers to disabling MSan for 72 projects due to enduring compatibility issues with instrumented libraries. Being MSan available only for LLVM, also software whose external dependencies (or the program itself) come built with *gcc* or other compilers is out of its reach. Furthermore, no UUM sanitization is available in

Network and Distributed System Security (NDSS) Symposium 2025
24-28 Febrary 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.241133
www.ndss-symposium.org

binary-level fuzzing systems that recently emerged for security analysis of COTS binaries [11], [12], [13], [14], a capability helpful for security assessment of proprietary products or software assembled with modules from third-party vendors.

We stipulate that these limitations do not stem primarily from implementation gaps, if at all. For example, while popular sanitizers for bounds checking inspect only memory load and store operations, UUM sanitizers must track the effects of all instructions and maintain initialization information also for intermediate values that do not reside in memory. This intuitively requires a careful modeling of program computations. In their study, the authors of MSan report how developers were often reluctant to use binary-level UUM sanitizers like Memcheck [15] and Dr. Memory [16] due to their high overheads, measured in the 10-20x range.

MSan successfully mitigated the problem by turning it into a simpler one, having the compiler both insert more efficient instrumentation and optimize the machinery for modeling instructions. However, this resulted in a less generally applicable solution. If we draw a comparison with the widely popular ASan, failing to instrument all code may result in false negatives for ASan (specifically, bugs affecting library implementations), and in a high false positives rate for MSan, a characteristic significantly more negative for the usability of a sanitizer tool [1]. We conclude that a UUM sanitizer that pursues generality must deal with the requirement of instrumenting all code *and* attain low overhead to be practical.

Our proposal: We present QMSAN, a fuzzer-friendly, practical, and performant sanitizer that detects UUM errors in binary code. As accurate detection of UUM errors is expensive, especially if done at the binary level, we propose an opportunistic multi-layer design to achieve efficiency.

For the vast majority of fuzzer executions, we analyze only memory load and store operations and check loaded bytes for potential violations. This design avoids the expensive *shadow propagation* instrumentation required on nearly all other program instructions for a UUM sanitizer to correctly handle legitimate copying of uninitialized memory. As a result, we call our reported violations *potential* as these may be false positives due to the incomplete modeling of program code.

However, when fuzzing, one may learn from the large plurality of executions conducted until that moment to assess the nature of violations. Based on this insight, if in an execution we reach a previously marked false positive, we simply ignore it. Conversely, for new violations, we repeat execution under full instrumentation and learn about their nature, memorizing details of false positives. Discriminating the identity of violations would normally require testing under full instrumentation or, worse, root cause analysis. As an efficient proxy, we propose to consider the address of the offending instruction and contextualize it to the execution to avoid the risk of treating distinct violations as identical. Our contextualization involves temporal (e.g., a priorly encountered violation) and spatial (e.g., the calling context) properties of the execution that are cheap to track and, in practice, discriminate well different violations occurring at a given address.

The design we propose dramatically reduces the need to perform slow executions under full instrumentation and yields appreciably faster end-to-end fuzzing performance. We devise it in the user emulation backend of QEMU, resulting in a solution, QMSAN, compatible with state-of-the-art fuzzers like AFL++ and able to analyze binaries from multiple platforms (we currently support the x64 and AArch64 architectures).

Our experimental evaluation estimates the bug-finding capability of QMSan on 10 OSS-Fuzz subjects without MSan support and 5 proprietary programs, exposing 44 new bugs. We responsibly disclosed them to the developers and cooperated in their mitigation; at the time of writing, 4 vendor-issued CVEs have been assigned. For performance analysis, we study the overheads of QMSan on MSan-compatible benchmarks used in prior studies on binary sanitization, measuring an overhead of 1.51x atop QEMU and a relative overhead of 1.55x on the slowdown that MSan's compiler-based instrumentation adds when fuzzing.

To summarize, we propose the following contributions:

- We conceptualize MSan's algorithm at the binary level for fuzzing purposes, addressing the key limitation of requiring instrumented libraries to make it practical.
- To achieve scalability, we design a multi-layer technique that reduces the slowdown without causing false positives, thanks to a fast opportunistic detector that applies accurate UUM analysis only to very few test cases.
- QMSAN: an open-source, multi-architecture implementation of the approach in the User emulation of QEMU, readily compatible with the AFL++ fuzzer. QMSAN is available at https://github.com/heinzeen/qmsan.
- We present experiments to estimate the bug finding abilities of QMSAN and measure accuracy and performance aspects of the approach, using a collection of 15 open-source and proprietary Linux programs. We identify and responsibly disclose 44 new bugs in 13 of them.

II. Background and Challenges

This section introduces techniques and challenges for detecting UUM errors, especially in combination with fuzzing.

A. Sanitizers and Fuzzing

Memory safety violations are the basis for the most severe security vulnerabilities and have been extensively researched [1]. As a first line of defense, programmers can look for problems in their code using tools based on static or dynamic program analysis.

Static analysis tools tend to "err" on the side of caution, being conservatively correct in exposing issues as they consider all possible program execution paths. Dynamic analysis tools, also colloquially "sanitizers", are more precise but work on a single execution instance at a time; they exist for memory safety errors, as well as for other classes of programming errors [17], [1]. In short, static analysis is prone to false positives, reporting potential issues that are not bugs, and dynamic analysis to false negatives, missing some bugs.

Sanitizers are currently in high popularity for their proven efficacy in vulnerability discovery [1]. To mitigate the coverage problem typical of dynamic program analysis, they can be used in combination with techniques designed to explore the execution space of a program. In particular, the combination with fuzzing has proven itself dramatically effective in practice, exposing many security-relevant vulnerabilities in heterogeneous code bases [18], [19], [20].

Sanitizers operate by instrumenting the application code with checks acting like tripwires. Traditional sanitizers like Memcheck [15] and Dr. Memory [16] use dynamic binary instrumentation to rewrite a program while it executes with machinery apt to intercept various memory safety problems, such as spatial or temporal violations of heap memory safety, and UUM errors that are the focus of this paper. However, binary-level sanitizers incur prohibitively high performance overheads for fuzzing [12], [11].

Compiler-assisted sanitizers like ASan and MSan reduce the overhead by inserting instrumentation during compilation. The key advantages to this approach are that the compiler can optimize and simplify or even remove redundant checks, has insights into memory and register layouts such as for stack frames, and the instrumentation can be more lightweight as the compiler does not operate on binary code. Nevertheless, security testing of binaries is important in many real-world scenarios due to lack of source code, and researchers have recently shown how to bring ASan-like capabilities to binary fuzzing [11], [12], [21], [13].

B. MSan

Initially developed for UUM bug detection at scale at Google, MSan [8] is the state-of-the-art sanitizer for UUM errors. It uses compile-time instrumentation in LLVM to augment program instructions with machinery to track and validate the initialization status of memory.

For tracking purposes, MSan maintains a *shadow memory* that tracks the initialization status of all program memory, which can change every time the program writes to memory. For validation purposes, MSan looks up the initialization status of the portion of the shadow memory corresponding to the memory the program wants to read. The shadow memory has the size of program memory and MSan efficiently indexes it by applying a bitmask to program memory addresses.

For the shadow memory to accurately reflect the effects of program computations, MSan must instrument *all* memory load and store operations. Let us consider a program fragment that copies data between two buffers: MSan must intercept and copy the initialization status of the source bytes, or a later read from the second buffer may wrongly appear as uninitialized.

Challenge 1*: UUM error detection requires tracking all memory accesses. Failing to do so leads to false positives.*

Another challenge in UUM sanitization is that detecting one or more uninitialized bytes in a load operation does not imply a UUM bug. With many code patterns, such as copying partially initialized buffers (or padding bytes) or optimizations in library code, reading and copying uninitialized data is a legitimate action [8], [15], [16]. This forces MSan, and similarly binary-level UUM sanitizers, to defer inspection to data uses instead of immediately reporting a violation at the read access. MSan checks the initialization status of data coming from memory when the program uses it for one of:

1) evaluating a branch condition involving such data;
2) dereferencing a pointer derived from such data;
3) passing such data as an argument to a system call or to a sensitive (e.g., involved in I/O) library call.

To track how read data flows into uses, UUM sanitizers have to track *all* types of instructions (except branches) and model for their effects: they must propagate the initialization status from the shadow memory throughout the execution until a data item sees its use or gets discarded. In essence, the sanitizer has to perform a forward information flow analysis.

In MSan, this mechanism is dubbed *shadow propagation* and follows predefined propagation rules. These rules fall into two kinds: approximate rules when coarse-grained modeling of the effects is acceptable for accuracy and desirable for performance, and precise rules if accurate modeling is cheap or when coarse-grained modeling would hamper propagation accuracy too much. As a result, shadow propagation avoids false positives from copying of uninitialized memory, a pattern that occurs fairly often in practice [8]. Unfortunately, shadow propagation appreciably increases run-time overhead for two reasons: the many more instructions it must instrument (besides load/store ones) and the variable modeling costs for them.

Challenge 2*: Accurate UUM error detection requires (i) tracking nearly all instruction types (to avoid false positives from legitimate copies of uninitialized bytes) and (ii) performing validation only at uses of memory-derived data.*

In light of *Challenge 1 and 2*, the attentive reader could argue that tracking memory accesses and subsequent computations may be necessary not only for program code, but for libraries, too. This is indeed true and applies to the C standard library already (unless the program uses from it only popular functions for which MSan provides an *interceptor*[1] that synthetically captures their execution effects). In our experience, failing to provide instrumented libraries results in an astounding number of false positives already after few minutes of fuzzing of programs of moderate complexity.

Challenge 3*: All library code must be instrumented, too.*

This third challenge impacts the compatibility and, therefore, the practicality of using MSan for fuzzing. For several programs, recompiling all libraries may be a daunting prospect or present additional challenges. We believe this fragility showed in the setback experienced by the OSS-Fuzz initiative

[1]This concept is present also in ASan to model the effects of common memory manipulation functions [12], mainly to avoid false positives with routines that allocate memory or manipulate it through highly optimized code.

```
1  void foo(){
2     char buf[4], a;
3     read(0, buf, 4);
4     a = buf[0];
5     if(a==MAGIC_BYTE)
6        puts("Hello world!");
7  }
```

```
1  push %rbp
2  movq %rsp, %rbp
3  subq $0x10, %rsp
4  leaq -0x4(%rbp),%rsi
5  xorl %edi, %edi
6  movl $0x4, %edx
7  call read
8  movb -0x4(%rbp),%al
9  movb %al, -0x5(%rbp)
10 movsbl -0x5(%rbp), %eax
11 cmpl 0x41, %eax
12 jne L
13 leaq hello_world, %rdi
14 call puts
15 L: addq $0x10, %rsp
16 pop %rbp
17 ret
```

Fig. 1. A simple function that reads from `stdin` and checks whether the first byte equals a predefined value. C source on the left, x64 assembly in the rest.

mentioned in Section I. Yet, our experimental results on a small sample of the OSS-Fuzz projects missing MSan testing suggest that many of them may harbor UUM errors that modern fuzzers would readily expose once given the "right" technical means.

C. Binary-level UUM Errors Detection

We use the fragment of Figure 1 to showcase some of the extra challenges that binary-level UUM sanitizers face compared to compiler-based solutions. These arise from multiple factors: information loss during compilation (e.g., storage location and lifetime of variables), optimized paths in library code that result in accesses that do not exist in source code (or in the compiler's intermediate representation), and the limited visibility binary sanitizers have on code (i.e., run-time only).

These factors cause binary-level UUM sanitizers to unnecessarily propagate shadow information and conduct superfluous checks, leading to higher run-time overhead. Fortunately, unlike address sanitization, these factors do not affect recall: that is, binary-level UUM detection does not face false negatives relatable to loss of information during compilation[2].

When MSan analyzes the C fragment on the left side of Figure 1, it instruments it as follows:

- On line 2, it marks as uninitialized the shadow memory portions corresponding to the storage for `a` and the contents of `buf`. We count this as two operations.
- On line 3, it marks the shadow memory for `buf[0]` depending on the outcome of the `read` operation. An interceptor aids the MSan runtime in modeling `read`.
- On line 4, it performs shadow propagation by copying the initialization status of the shadow memory for `buf[0]` into the shadow memory for `a`.
- On line 5, it checks whether `a` is initialized when evaluating the branch condition.

In total, MSan realizes a total of 5 operations from shadow memory initialization, shadow propagation, and initialization checking at uses. Thanks to the information available in the compiler intermediate representation, MSan omits unnecessary instrumentation for the three arguments for `read` (two are constants, whereas the compiler just defined the pointer to `buf`) and the one for `puts` (it can recognize it as a constant literal). Applying LLVM's `-O3` optimization to the code would then reduce the number of MSan operations to just 2: initializing the shadow memory for `buf[0]` and checking it at its use in the `if` statement after `a` is optimized away.

When a binary-level UUM sanitizer analyzes the assembly counterpart of the original code, it has to perform significantly many more operations. We describe the workflow below:

- Function prologue (lines 1-3) and epilogue (lines 15-17) require recognizing the allocation of 16 bytes on the stack (indistinguishably for `a` and `buf`), but also shadow propagation on the initialization status of `%rbp` during register spilling and fetching from stack.
- Argument preparation (lines 4-6) for calling `read` requires updating the initialization status for the *shadow registers* that the sanitizer tool must maintain alongside the shadow memory. Before allowing the call on line 7, the tool will inspect the three (shadow) registers that, according to the calling convention, carry the arguments for the sensitive API call at hand. The tool can then model the call with an interceptor to update the initialization status of `buf[0]` upon return.
- Evaluating the branch condition (lines 8-11) requires shadow propagation for shadow memory contents at `-0x4(%rbp)` and for shadow registers `%al` and `%eax`.
- Finally, the tool will do shadow propagation on line 13 and register inspection upon calling `puts` (line 14).

***Challenge 4**: Compared to compiler-based approaches like MSan, binary-level UUM error detection incurs higher overheads from unnecessary shadow propagation and checks.*

Finally, alongside having to analyze many more operations than compiler-based solutions, binary-level UUM sanitizers face additional challenges from optimized pathways in library code and other behaviors that access uninitialized memory without causing a semantic (or security) violation. We find the following cases to be the most common instances:

- *Stack probing*. Code reads from an uninitialized location to ensure stack growth before making an allocation that may overgrow the current page boundary.
- *Stack resizing*. Instead of adding to the stack pointer, compilers may shrink the stack by "popping" data from unused stack slots to dead registers. These operations can also occur alongside computations that move valid values back into registers for later use.

[2]The only exception we can think of are C++14's indeterminate values for expressions, which Memcheck handles with heuristics [1].

- *Aggressive optimization.* Libraries may feature behaviors such as load widening [9] to speed up memory operations, reading more data than strictly necessary (e.g., accessing 4 bytes for a 3-byte data element).

For example, in an execution of `/bin/ls` we measured:

Action type	Count	Location
Stack probing	2	libc
	40	/bin/ls
Stack resizing	20	/bin/ls
Load widening	151	libc

While stack probing may be easy to identify with heuristics, the other two cases demand the use of shadow propagation (or a similar dataflow analysis) to suppress harmless violations.

***Challenge 5**: Binary-level UUM error detection has to handle constructs that do not exist at the source code or the compiler intermediate representation-levels.*

III. Design

This section describes the design of QMSan by first presenting a general overview and then detailing its core components and how they face the five challenges of Section II.

A. Overview

QMSan combines the most desirable features of existing UUM sanitization approaches in a fuzzer-friendly design regarding both run-time overhead (the downside of binary-level tools like Memcheck and Dr. Memory) and compatibility (the downside of MSan) without reducing accuracy. QMSan not only enables UUM-aware fuzzing for binaries for the first time, but also offers a lightweight, plug-and-play alternative to MSan's deployment limitations when source code is available.

The five challenges of Section II shaped the three key components of QMSan's design: these synergistically cooperate to make UUM-aware fuzzing practical. We implemented QMSan atop QEMU's User Emulation for portability and compatibility with popular fuzzers such as AFL++.

The first component is an accurate binary-level UUM error detector. It inspects every program or library interaction with memory through load and store operations, and performs shadow propagation for all code until data sees a use (following the same logic of MSan, Section II-B) or is discarded, updating the shadow memory state in the process. Its capabilities are identical to those of existing binary-level UUM sanitizers.

The second component is an opportunistic UUM error detector and features the main technical novelty we propose in this paper. It integrates with a fuzzer and, during test case execution, it intercepts only load and store operations. When a memory read brings one or more uninitialized bytes, the component recognizes a potential UUM violation and saves it, allowing execution to continue. Upon test case execution completion, for each recorded violation the component checks the offending instruction and its context (temporal and spatial properties we detail in Section III-C) to see whether a similar violation turned out to be a false positive in the past. If new violations are found, it asks the accurate detector to process the test case, tracking which violations were a false positive, and informing the fuzzer if some violation was a true positive.

The third component aids both detectors with shadow memory management (when the program loads sections or manages heap memory) and interceptors. We use two types of interceptors: one to model the effects of system calls and another to ameliorate performance by further reducing how many violations the accurate detector component must analyze.

B. Accurate UUM Error Detector

This component reproduces in QEMU the working of state-of-the-art binary-level UUM detectors like Memcheck and Dr. Memory. It maintains a shadow representation for memory and for CPU registers, and oversees two tasks involving all the code both from the program and the libraries the program uses:

1) Inspecting interactions with memory contents and keeping the shadow memory status up-to-date;
2) Performing shadow propagation and check the initialization status of data upon relevant uses (i.e., branch conditions, memory addressing, sensitive APIs).

For both tasks, *Challenge 3* prescribes that instrumentation and analysis extend to instructions from program and library code to avoid otherwise frequent false positives. Dynamic binary instrumentation with QEMU can naturally achieve this.

For the first task, the component instruments all load and store operations (*Challenge 1*). For a load, it accesses and transfers initialization information for the memory source to the shadow representation it keeps for the destination register. For a store, if sourcing a register, it copies the register initialization status to the shadow memory for the destination, while with an immediate value it marks the memory as initialized.

For the second task, the component instruments all instructions for arithmetic-logic computations and non-local control flow transfers (*Challenge 2*). As we discussed in Section II-B, this task is responsible for most of the run-time overhead, as it must instrument and model the effects of these instructions.

Our design differs from MSan in what memory we track the status of. MSan tracks the initialization status only for objects for which either it witnesses an allocation and possibly a deallocation (like stack variables and heap objects) or their storage can be determined statically (like global variables). Doing this at the binary level would be prone to imprecision, as memory layout recovery is a hard problem in binary analysis; also, it is not necessary for UUM error identification. QMSan tracks initialization information for all memory, with a *default status* of uninitialized, and updates it upon store operations as described above. This choice incidentally enables QMSan also to expose addressability violations with invalid stack or heap addresses that MSan misses, as Section III-D will discuss.

We defer detailing shadow propagation rules to the implementation section. From a design perspective, we underline the role of shadow registers and the associated costs of tracking their initialization status (*Challenge 2*). In practice, only a small fraction of initialization state checks involves direct accesses to memory; very often, uses involve register contents derived from previously read memory values. For the sake of precision, QMSAN shadows general-purpose registers with up to bit-level precision for the initialization status of their data.

C. Opportunistic UUM Error Detector

The second component centers around the integration into the testing pipeline. Our key innovation is an opportunistic, multi-layer method to detect UUM errors when fuzzing that mitigates the performance penalties (shadow propagation and library code analysis) and compatibility limitations of existing binary-level and compiler-based methods.

Opportunistic Analysis: The main design choice we make for this component is to instrument only load and store operations, doing away with the expensive shadow propagation (*Challenge 2*) the accurate detector does on the other instructions. But as we noted in Section II-B, partial instrumentation promptly leads to false positives (*Challenge 1-2-3*).

However, in the context of a fuzzer, we stipulate that one can learn from the large plurality of executions conducted until that moment to assess the nature of violations. Based on this insight, if all the violations we find for a test case closely match violations that priorly turned out to be false positives in past executions, we ignore them; otherwise, we repeat execution through an accurate UUM error detector component and check the outcome, memorizing details of new false positives.

This filtering aims to avoid redundant validation work for false violations that are a consequence of our partial, propagation-free instrumentation. These may routinely surface, for example, every time the program runs a given operation on an object kind (e.g., copying a structure that contains padding bytes). Obviously, the filtering needs to be efficient enough so as to minimize the number of test cases that call for expensive validation, but also sufficiently precise to not discriminate two distinct violations as identical, or this may lead to false negatives. To efficiently and safely implement such a design, we therefore asked ourselves two questions:

1) *Do we have to validate violations in real time?*
2) *What defines the identity of a violation?*

Validating Violations: For the first question, we stipulate that violation assessment can, and should, be deferred. As we discussed in Section II-B, only data reaching relevant uses warrants sanitization, but tracking such data flows needs additional instrumentation (*Challenge 2*). However, in the presence of a UUM bug, any offending use necessarily derives from one of the (potentially many) data buffers reported as apparently uninitialized when loaded from memory.

Therefore, we can safely do away with real-time violation analysis and continue execution, minimizing the impact on the fuzzing throughput and allowing further violations to occur until execution ends: all violations will be processed together.

Then, to avoid thrashing effects between violations, we make the following choice. When handling a store operation, unlike the accurate detector where we copy the initialization status of the source, in the opportunistic detector we always mark the destination as initialized in the shadow memory. To justify the soundness of this choice, we note that if any written byte derives from uninitialized data, this detector would have anyway flagged the memory load operation that introduced said data, and will then assess whether to invoke its accurate counterpart to study the test case execution in its entirety.

In summary, we record all apparent violations during the execution of a test case, postponing their validation to the end. The curious reader may wonder whether, upon suppressing a violation, marking also the sourced memory as initialized would help. We think it would not: this could reduce analysis costs of future accesses on the involved bytes, but also lead to false negatives when a different violation (e.g., from another function) later occurs on one or more of the "silenced" bytes.

Identity of Violations: The ability to accurately recognize a reported violation as a previously encountered one is crucial for the accuracy and the scalability of our approach. We note that precisely determining if two violations are the same is intrinsically hard due to the factors at hand: the identity of the source object, the code accessing it, and the computations that follow the access until the use (which in turn depend on the program state when the loading takes place).

The characterization of violation identities has a large impact on accuracy and performance. A too loose definition may erroneously consider a violation as equivalent to one deemed as a false positive before: this may introduce false negatives, but incidentally higher throughputs thanks to the fewer test cases sent to the (expensive) accurate detector. On the other hand, an overly narrow definition may be costly to compute (e.g., it may need to closely track other execution aspects) and lead to frequent invocations of the accurate detector.

For the heuristics we explore in this paper, we chose not to take into account object identity. Accurate object identification is a hard problem in binary analysis. But even if accurate information were available, programs often create and destroy a large number of objects: precise identifiers would give us an overly sensitive criterion prone to explosion effects, whereas coarse-grained identifiers may pave the way to false negatives.

A first reasonable criterion to model a violation may be instead using the identity of the *offending instruction* that reads from memory that the opportunistic detector considers uninitialized. Instructions are trivial to identify and, most typically in practice, only a handful of them access uninitialized shadow memory regions across all program executions.

As we will see when discussing our experimental results, such a strategy already enables a fuzzer to expose UUM bugs, but may miss others depending on program and input characteristics. For example, at a given time, an instruction may turn out to be a load widening case (*Challenge 5*) or, more simply, an access to data that we capture imprecisely because we skip shadow propagation (*Challenge 2*). Once we add the instruction to an ignore list for suppressing violations, we will miss in the subsequent executions any UUM bug for which the instruction reads uninitialized data from another object, or even from the same object when initialized in a different way.

To tame this imprecision, we augment the offending instruction criterion with two heuristic feedback sources:

- **Spatial locality**. We further distinguish instructions by the calling context of the enclosing function. Context-sensitivity is a property known to enhance many program analyses. In our scenario, functions may work on different types of objects depending on the values flowing through different call graph ancestors to the enclosing function. Therefore, we use calling contexts as a partial, coarse-grained proxy for data-flow diversity [22], modeling them with a hash value.
- **Temporal locality**. When a test case raises multiple potential violations, their interleaving may be an additional source of information. We find a different sequence of violations indicative of the program engaging in different behaviors[3] than before. Starting with the second violation, we combine the identifier of the violation (instruction address, possibly augmented with spatial locality) with the one of the most recent violation. This provides us with a cheap way to capture different objects undergoing different manipulations that share some code during their processing work.

Our validation experiments show that, for the subjects we tested, the combination of the two heuristics does not cause any false negatives compared to MSan or binary-level sanitizers.

Library Instrumentation: The opportunistic detector supports selective instrumentation: users can configure it to ignore library-level memory accesses and track only program code for efficiency. If users want to focus on the program (a reasonable choice with well-tested libraries), QMSAN can skip library code analysis with a residual risk of false negatives.

When library code processes a program buffer with uninitialized parts, QMSAN may catch on such UUM errors when the program processes the output of the library. As QMSAN did not track the write operations that initialize such bytes, their shadow status appears as uninitialized and QMSAN forwards an apparent UUM violation to the accurate detector.

However, we foresee two cases evading this logic. In one, the output is a primitive value and the library does not write it to memory, as with return values of functions. In the other, library code receives one buffer containing uninitialized input and another buffer as the destination for UUM-affected computations; the program has written to the destination buffer before (therefore, in future reads, QMSAN sees a stale initialization status from those writes) and reuses it here.

In various tests, we observed instances of the first case with C library functions that analyze buffers, as with comparators, and apparently no instances of the second case. Therefore, for the evaluation, we enable selective instrumentation and use standard interceptors for such functions. In summary, QMSAN's design tackles *Challenge 3* and, partially, *Challenge 4-5* for burden ascribable to libraries, whereas all existing solutions would easily incur false positives (and false negatives) if they ignore library code.

[3]Drawing an analogy with coverage-guided fuzzing, this is similar to using edge rather than basic block identity to recognize novel interesting behaviors.

D. Runtime Helper Component

The third component of QMSAN assists its two detectors in additional tasks. As we mentioned, one is to update the shadow memory to account for statically initialized contents from an executable and for dynamic heap allocation.

Free operations are particularly important for UUM detection because the allocator may reassign a buffer to a subsequent allocation operation. Therefore, QMSAN must mark any freed bytes as uninitialized or it would miss UUM errors from an incomplete initialization of a new object that reuses storage.

As anticipated in Section III-B, this also allows us to detect accesses to a previously freed buffer and other rogue memory dereferencing on stack or heap. Technically speaking, these are addressability bugs that MSan misses[4], but our design catches them for free as we must track allocations for correctness. Section V-C will detail some exemplary bugs we found.

The other main task we delegate to this component is devising interceptors. We distinguish between those needed for correctness and others that ameliorate performance. The first kind model the output of system calls that allocate or populate buffers on behalf of user code: this necessity is shared with existing UUM sanitizers, as otherwise one would all miss the effects of memory manipulations from kernel code.

The second kind of interceptors aid both detectors. We intercept the invocation of C library functions for memory and string manipulation with highly optimized implementations (e.g., load widening, inline assembly sequences, or vectorized alternatives). For the accurate detector, these are not strictly needed, but ease the implementation work for exotic cases. For the opportunistic detector, modeling them can reduce the total amount of unique violations requiring accurate analysis that originate from invoking a few functions of common use.

E. Discussion

Our design realizes a practical approach to UUM error detection when fuzzing, limiting the use of more expensive operations (like shadow propagation) to when previously unseen violations occur. New violations present themselves rarely: in our experiments, 99.8% of executions are opportunistic (Table VII). The net effect of using only the lighter opportunistic detector for most executions is enhanced performance, which in turn can lead to strong practical value as we will show by testing heterogeneous code bases harboring UUM bugs.

The technical innovation of QMSAN lets us handle off-the-shelf binaries for fuzzing (a scenario precluded to existing binary-level tools due to their overheads) and overcome practical limitations of MSan that hinder its use for fuzzing (mainly, the need to instrument through recompilation all program and library code to avoid otherwise frequent false positives).

[4]Its authors explain in [8] that proper handling of (general) addressability bugs introduces complexity (e.g., maintaining a richer shadow map and adding red zones), and that running ASan after MSan is still faster than using binary-level sanitizers able to capture both kinds of errors. The paper was published before fuzzing in combination with sanitizers became a popular combination.

In terms of soundness, our design avoids false positives: each potential violation the opportunistic detector spots for the first time undergoes inspection in the accurate detector, which is as sound as existing UUM tools (Appendix A). However, by suppressing violations that closely resemble priorly discarded false positives, our approach may incur false negatives if violation identification is imprecise (or with the edge cases of Section III-C when skipping library code). While we did not observe false negatives in our accuracy tests, this does not imply their absence in other settings. We believe residual risk roots in two aspects: rare patterns (e.g., uninitialized data flowing through multiple levels of memory indirection) and nondeterministic execution. Future work may study the characteristics and prevalence of the former and propose new refinement heuristics. The latter is instead mitigated when the fuzzer later generates similar test cases for execution.

For portability, our design requires standard binary instrumentation capabilities. We believe one may bring it to Windows systems with modest implementation work, likely atop DynamoRIO [23] for ease of integration with WinAFL and derived fuzzers. As Windows supports only x64 and AArch64 platforms, an implementation shortcut may be to use an existing sanitizer (e.g., Dr. Memory) as accurate detector.

A promising avenue that our design enables is the concurrent use of (binary-level) ASan and MSan-style sanitization when fuzzing. Currently, users have to compile and test two separate binaries due to runtime incompatibilities. In preliminary experiments, we verified that our additions to QEMU seamlessly coexist with those of QASan. In future work, we would like to optimize their combination by having them share the shadow memory and relevant instrumentation code.

To further optimize the opportunistic detector, one may use static analysis to identify and ignore load operations on bytes guaranteed to be initialized (*Challenge 4*). This may be the case, for example, with stack reads dominated in the control flow by store operations on those locations. A static analysis to use for this end must come with soundness guarantees or the optimization could cause false negatives. As binary-level sanitizers have visibility on code only when about to execute it (Section II-C), such an optimization would be orthogonal to the design, but we would like to explore it in future work.

IV. Implementation

This section details relevant implementation aspects of QMSan. We add ∼5000 LOC atop the User Emulation component of QEMU (v. 3.1.1), which AFL++ uses as a high-performance binary fuzzing backend. Among them, we use ∼1500 LOC to implement 70 TCG-level lifters to manage shadow propagation and ∼1800 LOC to implement 110 interceptors. We then add another ∼300 LOC to AFL++ to carry the violation analysis logic at the end of test case execution, issuing an execution under the accurate detector when needed and tracking the known false violations.

The shadow map mirrors the 64-bit address space with a 32 TB zero-initialized memory buffer allocated at program startup: thanks to standard operating system optimizations,

V = initialized
V = uninitialized
	V' = 0xff
	V' = 0

Reg = imm
Reg_1 = Reg_2
Reg = load Mem
Mem = store Reg
Mem = imm
	Reg' = initialized
	Reg_1' = Reg_2'
	Reg' = Mem'
	Mem' = Reg'
	Mem' = initialized

V = A & B $V = A \mid B$ $V = A \oplus B$ $V = A \ll B$	$V' = (A' \mid B') \& (\sim A \mid B') \& (A' \mid \sim B)$ $V' = (A' \mid B') \& (A \mid B') \& (A' \mid B)$ V' = A' & B' $V' = \begin{cases} (A' \ll B) & \text{initialized(B)} \\ 0 & \text{uninitialized(B)} \end{cases}$
V = math(A, B)	V' = A' & B'

TABLE I
SHADOW PROPAGATION RULES FOR INSTRUCTIONS. GIVEN A PROGRAM VALUE `VAL`, `VAL'` IS ITS SHADOW-REPRESENTATION VALUE. FOR THE SAKE OF BREVITY, WE REPRESENT REGISTERS HERE AS A WHOLE; QMSAN MANAGES THEM WITH UP TO BIT-LEVEL PRECISION.

only a (program workload-dependent) fraction of it will result in allocating physical pages. This choice is analogous to MSan. However, to account for our different treatment of storage lifetime, we flip the meaning of the 0 value: that is, 0 means initialized for MSan and uninitialized for QMSan. Except for statically populated regions (e.g., `.rodata`), QMSan considers all memory uninitialized as default.

For the accurate detector component, Table I lists the propagation rules of MSan that we adapted to account for the different value semantics and for modeling shadow registers. Following the implementations of existing UUM sanitizers, for propagation we balance between precise and coarse-grained modeling of instructions. Operations in the third table group need only a few instructions to be modeled with bit-level accuracy: this is fortunate, as this accuracy removes otherwise recurrent false positives with string manipulations. For the other operations, as precise bit-level reasoning would be too expensive, we use a fast instrumentation (last table row[5]) to propagate the presence of any uninitialized bit in either operand to the destination, a choice done also by other tools. Our implementation also supports vectorized instructions.

For the opportunistic detector component, we mentioned that QMSan records all violations and inspects them only at the end of the execution, looking for new violations that warrant analysis in the accurate detector. Conceptually, this is similar to a coverage-guided fuzzer that tracks coverage information through coverage map updates and only when execution ends it inspects the test case's map against a global map looking for newly met coverage. Our implementation operates similarly: we maintain a global "violations map" across executions and one for the current test case. If the latter contains one or more previously unseen potential violations, we feed the test case to the accurate detector component and examine them, updating our global map to memorize violations that turned out to be a false positive.

Finally, for the helper component, we use the `LD_PRELOAD` method to supply to the program under test a library that interposes on selected functions: specifically, those for heap management (e.g., `malloc`, `realloc`, `free`) and those for which we want to register interceptors. As mentioned, we use

110 interceptors: we borrowed ∼40 of them from QASan [12], and the rest are original. As a rough proxy of implementation complexity, MSan uses 106 own interceptors and 601 shared with other sanitizers for LLVM.

V. Evaluation

This section presents the experimental results we obtained from tests conducted to estimate the bug finding capabilities of QMSan on real-world software and to analyze accuracy and run-time overhead aspects of the design. We also conduct component analysis studies to better understand its performance.

The most appreciable end-to-end outcome for software testing is, obviously, the number of bugs found. As much software is currently out of reach of MSan for fuzzing, either for compatibility struggles with whole-code instrumentation or for the unavailability of the source code, we assemble a pool of test subjects drawing from open-source projects and commercial software. Following the best practices in the fuzzing literature (e.g., [24]), we opt for popular programs considered in studies related to ours and, where applicable, vary one evaluation dimension at a time. The online repository for QMSan hosts the key materials of the conducted evaluation.

All the tests took place on a machine equipped with an Intel Xeon E5-2699 v4 CPU with 44 physical cores running at 2.20GHz and 256 GB of RAM with low background activity. We bound every trial to a physical CPU core, leaving its virtual counterpart idle and using a Docker container with Ubuntu 20.04. We use AFL++ 4.10 with deferred initialization to optimize the fork server[6] performance and leave parameters for fuzzing techniques at their defaults. For the test involving open-source subjects, we use clang 18 for compiling binaries.

To avoid unnecessary repetitions in the following sections, we remark that all the violations that QMSan found in our tests were true positives, according to consistency checks we did in Memcheck, in MSan if applicable, and manually. All the bugs we reported to developers were confirmed. To summarize, as also expected by design, we experienced no false positives.

Our experiments aim to tackle four research questions:

- **RQ1**: Can QMSan expose UUM bugs we are missing in otherwise well-tested open-source software?
- **RQ2**: Can it do the same for proprietary software?
- **RQ3**: How accurate is QMSan? Does it miss behaviors compared to source-based instrumentation?
- **RQ4**: What run-time overhead does QMSan incur? What are its sources and on what does it save?

Responsible disclosure: We followed community best-practices to disclose all the bugs we found during our tests to the relevant parties. We contacted project maintainers, following their prescriptions (e.g., `SECURITY.md` for GitHub projects and guidelines from company websites) when available, and proposed a timeline of 90 days before reporting any of our findings to the public. We offered our availability to provide additional insights on the bugs and general UUM issues; when our understanding of the code base allowed, we also proposed concrete fixes. Incidentally, we engaged in rich and constructive discussions with some of the maintainers.

[5]Due to the zero semantics, & yields 0 if either or both bits are uninitialized.

[6]As Appendix B discusses, QMSan can work also with persistent fuzzing.

Subject	Version	Fuzzing harness	Bugs	Fixed
libredwg	763d702	llvmfuzz	3	2
gpac	205bfe3	fuzz_probe_analyze	1	1
assimp	b71b8f7	assimp_fuzzer	2	1
libdwarf	6178ba8	fuzz_debug_str	2	2
serenity	7914383	FuzzJS	1	1
opensc	fe2c1c8	fuzz_pkcs15init	5	5
ntopng	8786f06	fuzz_dissect_packet	1	1
upx	3495d1a	test_packed_file_fuzzer	2	2
radare2	cfe5806	ia_fuzz	0	0
libucl	5c58d0d	ucl_add_string_fuzzer	0	0
Total			17	15

TABLE II
Dataset of OSS-Fuzz subjects used to estimate QMSan's bug finding capabilities. The table also shows how many bugs, at the time of writing, have been fixed following the responsible disclosure we did with their developers.

A. RQ1: UUM Bugs in Well-Tested Open Source Software

As we mentioned earlier in the paper, only about 2 every 5 subjects tested daily by the OSS-Fuzz initiative (40.5%) see MSan support, a situation stemming from compatibility struggles for the others with the requirement of having to instrument all libraries. Therefore, as a first experiment to estimate the bug finding capabilities of QMSan, we target open-source projects that are routinely tested by the community but lack explicit means to expose UUM bugs (if not as a consequence of direct crashes or other sanitizer-found errors in surrounding code).

To identify a reasonably small yet significant subset of OSS-Fuzz C/C++ projects suited for this evaluation, we inspected the OSS-Fuzz bug tracker and selected the top 10 subjects unsupported for MSan with the highest `bug-security` counts for bugs that OSS-Fuzz spotted in the 15 months (January 2023 to March 2024) preceding our experiments. As we want to estimate how QMSan finds bugs in otherwise well-tested software, our rationale is that higher bug counts indicate extensive previous testing for OOB/UAF errors (hence the tag) and spontaneous crashes. The projects are popular and their maintainers typically act on each reported bug readily, an attitude that showed also in our interactions. OSS-Fuzz has been testing most of these projects for 3 to 6 years, with the exception of the industry-strength `ntopng` network traffic probing tool and the popular `upx` executable packer.

Table II provides the list of subjects and the manually deduplicated, previously unknown UUM bugs we found in them. The subjects appear in descending order by the selection criterion[7]. Among the subject's harnesses available on OSS-Fuzz, we chose the one that was behind most `bug-security` issues reported in the considered time range. We used the

[7]We remark that we had to scroll through the initial top-10 as `ruby`, `http-pattern-matcher`, and `clamav` faced a temporary issue in their OSS-Fuzz build script due to upstream updates, and `gdal` hit an instrumentation error in the TCG component of the QEMU version we use for QMSan.

seeds and dictionary provided by OSS-Fuzz when available, and none otherwise. We conducted 3 fuzzing trials of 72 hours each, exposing 17 bugs among 8 of the 10 subjects; the developers of `opensc` issued CVE-2024-45616 for the bugs.

Most bugs were triggered by either corner cases in object initialization or bad error management. For example, in the `gpac` multimedia framework, a size variable was initialized with data from the input's header field; this size controls a heap allocation of a buffer for storing part of the input. By providing a controlled size in the input archive header, an attacker could control the size of the allocation. Providing a size value bigger than the input size resulted in a buffer only partially initialized, and this data would later undergo a decompression algorithm.

We also found an unusual bug in the JavaScript library of SerenityOS, detected for a buffer that the program passes to `setjmp`. The POSIX standard does not prescribe whether `setjmp` has to initialize the signal mask in the context data it writes to the buffer, and different versions of Linux and BSD operating systems make different choices for it. The documentation of SerenityOS did not cover this point. Shortly after in the execution, a function loops over these indeterminate bytes and conducts non-trivial operations within the `Heap` module of the library. When we reported the potential issue, the developers fixed it by zero-initializing the buffer.

Finally, we received a valuable comment by a developer of `opensc`, a set of tools and libraries for working with smart cards with cryptographic capabilities. The developer mentioned that the issues we reported differed from what OSS-Fuzz exposed over the last couple of years, and asked us *"if there is something we would propose to implement to allow OSS-Fuzz to progress further and detect other issues"*.

We found this comment to reflect the motivation that we argued for QMSAN at the beginning of this paper. In light of the promising results for bug finding ability and accuracy, our intention would be to eventually discuss with the OSS-Fuzz maintainers a potential integration of QMSAN in the pipeline.

B. RQ2: UUM Bugs in Proprietary Software

While a compiler-assisted solution (though with key limitations) for UUM-aware fuzzing exists, software available only in binary form cannot undergo this kind of testing. To estimate how QMSAN can aid security analysts, as well as developers that receive third-party components from external sources for integration in their products, we conduct experiments on a collection of real-world, closed-source software. We consider 5 subjects and a total of 9 binaries, testing two versions of a program when the first bugs we found and reported were fixed for a later version (this happened for all but one subject).

To assemble the subject selection, our options were more limited than in the open-source scenario. We reviewed recent papers on binary-level fuzzers or binary rewriters that can support fuzzers [25], [13] looking for candidates that could take input from the command line. We made this choice because writing a harness for a closed-source program is an orthogonal effort that faces practical complexity and potential license issues in the required reverse engineering activity: these are unrelated to the goals of our paper. This holds especially for programs with a graphical interface, which appear to us more numerous in the proprietary domain than command-line tools, as testing programs that expect user interaction is a hard problem for fuzzing [26]. Therefore, we came up with the 5 subjects listed in Table III. Compared to [25], although it syntactically meets our criteria, we leave out `lzturbo` as the program has not received updates since 2014: in a feasibility 3-hour fuzzing trial, QMSAN found 15 unique crashes in it.

Unlike the experiments for RQ1 where we could inspect sources to triage bugs, here we initially resort to stacktrace-based crash deduplication using the top 3 frames on the stack. Then, we refine the candidate bugs with an inspection of the involved instructions, the object addresses, and their surroundings. Upon reporting the candidate bugs to the developers, we asked them to confirm or correct our proposed bug counts.

The UUM bugs we list in Table III are the overall result of 3 fuzzing trials of 72 hours each. We did not provide dictionaries to AFL++. As seeds, we used a PNG image from the default seeds of AFL for `nconvert` and `pngout`, a minimalistic RAR archive for `rar`, and a tiny ELF files of 100 bytes for `cuobjdump` and `nvdisasm`. In total, we found 27 bugs.

The tests took place in two phases. Initially, we downloaded the latest program version available from official sources and tested it. For the two binaries from the NVIDIA CUDA toolkit, QMSAN found 2 bugs in `cuobjdump` and 7 in `nvdisasm`. NVIDIA promptly confirmed the bugs, issuing CVE-2024-0072 and CVE-2024-0076 and fixing them in the next release. For `nconvert`, we found 5 bugs confirmed by the maintainer. For `pngout`, the maintainer was skeptical on the usefulness of randomly-looking test cases, claiming the detection logic for most invalid images worked reasonably well; for the other bug, they acknowledged deeper implications, but showed no resolve in fixing it soon. Finally, for `rar`, the maintainer conducted a thorough, prompt investigation of the matter, triaging the bug to the use of an uninitialized HMAC key. The developer concluded that, to have security implications, one must produce a valid authentication code from the wrong source data, but the collision probability is unaffected by the initialization status of the key; we agreed. The anti-pattern was fixed shortly after.

A few months later, we tested the new program versions that became available in the meantime. We found another 3 bugs in `nvdisasm` (with NVIDIA issuing CVE-2024-0102), 4 in `nconvert` (two had been fixed in the internal development version), and 3 in `rar`. The RARLAB developers promptly got back to us with detailed security analyses. One bug emerged during their internal testing too and was cosmetic. Another bug could lead to a corrupt archive depending on the memory state, but broken files would be deleted after extraction due to checksum mismatches. Finally, the third bug could be used for denial of service on Linux clients. We requested MITRE to issue a CVE identifier and the case is being evaluated. All these bugs have been fixed at the time of

Subject	Vendor	Version	Bugs	Fixed
cuobjdump	NVIDIA	12.3	2	2
cuobjdump	NVIDIA	12.4	0	0
nconvert	XnView Software	7.136	5	5
nconvert	XnView Software	7.155	4	4
nvdisasm	NVIDIA	12.3	7	7
nvdisasm	NVIDIA	12.4	3	3
pngout	Ken Silverman	Jan 15 2020	2	0
rar	rarlab	6.11	1	1
rar	rarlab	7.0	3	3
Total			27	25

TABLE III
DATASET OF PROPRIETARY SOFTWARE USED TO ESTIMATE QMSAN'S BUG FINDING CAPABILITIES. THE TABLE ALSO SHOWS HOW MANY OF THEM HAVE BEEN FIXED AT THE TIME OF WRITING.

	Unique crashes		Component analysis		
Subject	MSan	QMSAN	Address	A+T	A+T+S
c-ares	0	0	0	0	0
guetzli	1	1	1	1	1
json	0	0	0	0	0
libxml2	2	27	15	19	22
openssl	0	1	1	1	1
pcre2	1	3	1	3	3
re2	1	2	1	1	2
woff2	0	0	0	0	0
Total	5	34	19	25	29

TABLE IV
CRASHES FOUND ON FTS SUBJECTS. 'A' STANDS FOR ADDRESS; 'T' AND 'S' FOR TEMPORAL AND SPATIAL, RESPECTIVELY.

writing.

C. RQ3: Accuracy Aspects

The bug finding results for the experiments tackling RQ1 and RQ2 provide a meaningful, yet partial picture of the efficacy of QMSAN. We designed a multiplicity of experiments to assess accuracy aspects of our approach in depth.

Juliet: For the accurate detector, besides tests on commodity Linux command-line programs that we ran in lockstep with Memcheck, we use the Juliet Test Suite, released by NIST and containing programs with various weaknesses. Specifically, we focus on the weakness type pertinent to our work: Use-of-Uninitialized-Variable (CWE-457). Juliet comes with about 1000 test instances for it: those exercise UUM errors on multiple data types, with either partial or no initialization at all, and involving stack or heap memory. Each test comes in two flavors: a good and a weak (i.e., buggy) version of the action of interest. Some tests have nondeterministic features (e.g., they depend on the output of the `rand()` function) for assessing static analysis tools: therefore, similarly as in [12], we identify and remove them from the pool. On the 884 tests that remain, we obtain perfect accuracy for our accurate detector. As for the opportunistic detector, it recognizes all true positive cases, and misjudges 20 true negatives as false positives: in a fuzzing setting, all those 20 instances would be sent to the accurate detector for inspection, and the opportunistic one would learn about their nature from the other's output and memorize them.

MSan-compatible Subjects: A more demanding test than Juliet is testing the accuracy of QMSAN against the output of MSan. Therefore, we identify a pool of MSan-compatible programs that we can use to answer RQ3 and, later, also RQ4.

Our choice falls on 8 programs from the Google's Fuzzer Test Suite (FTS) considered in notable recent literature on sanitization, especially in combination with fuzzing [12], [19], [20], [27]. The 8 subjects coincide with those from the evaluation of QASan [12] against ASan and, most notably for our purpose, need only the C library to be MSan-instrumented to avoid false positives. We choose their FTS version as our goal is to compare bug recall ability: most typically, older program versions feature more bugs than more up-to-date counterparts.

In the left part of Table IV, we report the overall results of 3 fuzzing trials of 24 hours each where we compare QMSAN with MSan. When available, we use seeds and dictionaries from FTS and, as second choice, OSS-Fuzz. QMSAN finds all the 5 unique crashes that MSan does, but also another 29: 25 in `libxml2`, 2 in `pcre2`, and 1 in `openssl` and `re2`. All the crashes were induced by the tripwires of the sanitizers (i.e., the program would not have crashed spontaneously).

From a close examination of the crashes and the respective fuzzer queues, all the additional `libxml2` crashes traverse CFG edges that AFL++ did not reach when testing the compiler-instrumented binary. While many of these crashes may be related (we see they all insist on a buffer hosting a copy of the input), a coverage-guided fuzzer uses code coverage to discriminate differences in execution, and only post-mortem manual analysis can precisely triage bugs. Therefore, any additional input that exposes a UUM error from a different instruction (or stacktrace) is potentially valuable.

For the other bugs that MSan misses, the analysis is as follows. For both `openssl` and `re2`, the additional bug comes from better exploitation of code reached by both QMSAN and MSan. For `pcre2`, both bugs are addressability issues that only our design can capture (Section III-D).

Component Analysis: To understand how the temporal and spatial locality refinements presented in Section III-C contribute to make our approach precise, in the right part of Table IV we report the results of an ablation study by repeating the FTS experiments in a modified QMSAN version. The numbers are collected on the same fuzzing campaign, thanks to debugging controls that allow us to analyze potential violations separately for each assortment of heuristics. Column *Address* reports the unique crashes due to UUM violations identified by considering only the program counter of the offending memory load instruction. Column *A+T* reports data for when we refine the criterion with temporal locality. Column *A+T+S* shows the fully-fledged approach where we also introduce spatial locality. In these tests, the temporal heuristic exposes more bugs on `libxml2` and `re2`, and the general results improve appreciably when also adding the spatial one. In retrospective, when we designed the heuristics and preliminarily tested them with a larger pool of programs, we noted a slightly more pronounced impact from the temporal heuristics. Nevertheless,

	Native	MSan	QEMU	QMSAN			QMSAN[†]	
Subject	exec/sec	vs. Native	vs. Native	vs. Native	vs. MSan	vs. QEMU	vs. QEMU	vs. QMSAN
c-ares	2053.00	2.09x	2.11x	2.20x	1.05x	1.04x	1.41x	1.35x
guetzli	2096.14	2.56x	2.25x	3.17x	1.24x	1.41x	2.84x	2.02x
json	1747.18	2.17x	2.42x	2.69x	1.24x	1.12x	2.55x	2.28x
libxml2	1427.36	3.79x	2.40x	3.41x	0.90x	1.42x	3.81x	2.69x
openssl	362.51	2.41x	4.24x	19.84x	8.24x	4.68x	17.48x	3.74x
pcre2	1846.23	2.24x	2.28x	3.18x	1.42x	1.40x	1.75x	1.25x
re2	1788.10	2.27x	2.26x	3.35x	1.48x	1.48x	2.34x	1.57x
woff2	2127.90	2.13x	2.37x	2.86x	1.34x	1.20x	2.37x	1.97x
geomean	-	2.41x	2.48x	3.75x	1.55x	1.51x	3.00x	1.99x

TABLE V
FTS SUBJECTS: THROUGHPUT FOR AFL++ WITH COMPILER-BASED INSTRUMENTATION, MSAN INSTRUMENTATION, QEMU INSTRUMENTATION, AND WITH QMSAN AND QMSAN[†]. RELATIVE OVERHEADS ARE GIVEN FOR THE EASE OF COMPARISON.

we can conclude that the component analysis shows how both refinements contributed to precise violation identification in the subjects we tested here, putting QMSAN on par with (or even above) the "accurate-by-design" results obtained by MSan.

Additional Tests: We conclude our examination of RQ3 by detailing additional experiments we performed on QMSAN.

We first focus on the accuracy of QMSAN compared to existing binary-level UUM sanitizers. First, for all QMSAN queues available at the end of the experiments conducted for RQ1, RQ2, and the parts of RQ3 discussed above, we individually execute in Memcheck and Dr. Memory all the test cases in which QMSAN found no violations. None resulted in a UUM error, which would be indicative of a false negative.

Then, we restart QMSAN from a selection of these queues and configure it to record the first 100 000 test cases AFL++ generates. The rationale is that a test case may feature new UUM behaviors without bringing new code coverage, so a fuzzer would not save it; the prior experiment missed this. Restarting from saturated queues allows us to record inputs that are reasonably more complex than those a fuzzer would generate from simpler seeds. Existing tools raised no UUM errors when processing these new test cases (details in Appendix A).

Finally, while our evaluation is centered on the x64 architecture for practical reasons, we made preliminary experiments on QMSAN's AArch64 backend to test its maturity. Alongside validation experiments with command-line programs (both in the accurate detector alone and in the whole system), we repeated the RQ2 experiments on the two proprietary programs that are available also for AArch64: `cuobjdump` and `nvdisasm` from the NVIDIA CUDA toolkit. While these binaries are naturally different from their x64 counterparts, QMSAN was able to expose the same bugs as it did for the latter, also with no significant difference in their time to exposure. Albeit partial, we find these tests may be indicative of good practical value that follow-up work can capitalize on.

D. RQ4: Performance Aspects of QMSAN

We conclude our experimental analysis by examining performance aspects of QMSAN: run-time overheads, its sources, and the dimensions on which it can effectively save thanks to its design. We start by discussing a key metric for fuzzer performance: its throughput, commonly measured in terms of test case executions completed per time unit (seconds).

Table V shows the relative slowdowns of different configurations on the FTS subjects we used for tackling RQ3. We plot the median value from a series of five 24-h trials. We opt for these subjects so we can compare the overheads of QMSAN also against MSan, as well as for consistency with the accuracy considerations we provided in the previous section. In the column *Native*, we report the number of executions per second that AFL++ achieves when using its performant `afl-clang-fast` backend with compiler-assisted instrumentation via LLVM. Next, we provide relative slowdowns for MSan, for binary-level fuzzing using the QEMU User Emulation backend of AFL++, and for UUM-aware fuzzing with QMSAN. In this section, we also introduce a new fuzzer configuration, QMSAN[†], which is a version of QMSAN modified to execute every test case in the accurate detector (i.e., we ablate the contribution of the opportunistic component).

The first result that we observe by looking at general trends, captured by the geometric mean of the slowdown ratios, is that QMSAN halves (1.99x speedup) the run-time overhead that shadow propagation and library code analysis would introduce in the execution. In practical terms, with QMSAN, AFL++ could test on average twice as many test cases than with QMSAN[†] in the given 24-hour budget. Then, compared to a UUM-unaware fuzzing campaign in QEMU, we measure an average slowdown of 1.51x. For a rough comparison, MSan introduces a 2.41x slowdown atop the compiler-based instrumentation of AFL++. These numbers suggest that our approach is particularly efficient in terms of fuzzing throughput, obtaining a performance drop of 1.55x compared to MSan despite the latter can benefit from a much more optimized backend (under QEMU, AFL++ is on average 2.48x slower than with compiler-based instrumentation).

Only on `openssl` we incur higher overheads. We will return to this subject later in the section, as we will discuss the shadow memory access patterns we measure for our subjects. Interestingly, in additional tests we measured substantially lower slowdowns with its other two harnesses from FTS; the one we use here is the only one compatible with MSan.

	QMSan$^-$	QMSan		QMSan†	
Subject	vs. QEMU	vs. QEMU	vs. QMSan$^-$	vs. QEMU	vs. QMSan
c-ares	1.12x	1.20x	1.07x	1.45x	1.20x
guetzli	1.40x	1.74x	1.24x	6.80x	3.90x
json	1.20x	1.24x	1.03x	3.40x	2.75x
libxml2	1.45x	1.76x	1.21x	5.84x	3.32x
openssl	3.62x	3.96x	1.09x	16.87x	4.27x
pcre2	1.22x	1.36x	1.11x	3.48x	2.57x
re2	1.31x	1.52x	1.16x	4.38x	2.88x
woff2	1.55x	1.70x	1.10x	4.66x	2.74x
geomean	1.49x	1.68x	1.13x	4.69x	2.79x

TABLE VI
RELATIVE SLOWDOWN FOR TEST CASE PROCESSING TIME.

Subject	Total executions	Requested analyses	One every
c-ares	80 633 233	0	-
guetzli	57 145 827	48	1 190 538
json	56 049 871	2 901	19 321
libxml2	36 183 734	926	39 075
openssl	1 578 914	8	197 364
pcre2	50 169 305	8 716	5 756
re2	46 053 114	2 110	21 826
woff2	64 339 311	9	7 148 812

TABLE VII
FTS SUBJECTS: NUMBER OF EXECUTIONS AND ISSUED REQUESTS FOR ACCURATE ANALYSIS DURING 24 HOURS OF FUZZING.

Fine-grained overhead analysis: We refine the performance considerations above with additional experiments for two reasons. First, besides entropy reasons, fuzzers using different instrumentations may slightly diverge in the program behaviors they explore, as different instrumentations can affect, for example, what test cases the fuzzer prioritizes as favored. Second, end-to-end measurements can lead to underestimating the performance contributions, because they include the costs of input mutation and queue management after test execution.

To establish a more refined ground for studying the impact of our choices during binary-level fuzzing, we pick for each subject $100,000$ test cases from a saturated corpus (just like we did for RQ3 in the previous section) and run them in all QEMU-based fuzzers. Table VI displays the relative slowdowns in test case processing time. In this experiment, we also analyze another ablated configuration for QMSan: specifically, QMSan$^-$ is designed to treat all memory as always initialized, hence it ablates from QMSan the costs of updating and checking the map of potential violations.

We can observe that violation assessment slows down QMSan by a 1.13x factor, enabling by design a much higher performance gain. While we could measure an end-to-end performance gain of 1.99x from QMSan over QMSan†, these tests show a real benefit of 2.79x, measured by having both fuzzers analyze the same test cases. We find that these numbers, adding to the end-to-end overheads we discussed above, further substantiate the efficiency benefits that we claim for our multi-layer, opportunistic design behind QMSan.

Other measurements: We conclude the treatment of RQ4 by detailing data points for two relevant phenomena.

Based on the runtime overheads we discussed above, the reader may be tempted to think that our design results in a modest number of invocations of the accurate detector component during an execution. As our readers may see in Table VII, we request very few test case executions during a fuzzing campaign (and even never on `c-ares`). We can also safely conclude that these requests are not the reason behind the higher overheads we mentioned for fuzzing `openssl`, as we issue 8 requests in total (approximately once every 200K completed executions, which amount to 1.6M in 24 hours).

Finally, we provide figures we collected on the shadow map accesses to provide an additional perspective on the performance gains of our approach. For several subjects, these reflect what we anticipated in *Challenge 4 and 5* for how binary-level approaches are at a disadvantage compared to source-based solutions due to the differences in visibility and in the code constructs. The figures we collect do not capture shadow propagation or library instrumentation costs, but focus on load and store operations from program code, as even our efficient opportunistic design must instrument completely. Each of them necessarily causes a shadow memory access.

Table VIII shows the data for the FTS subjects with $1,000$ test cases picked uniformly at random from the larger pool of $100,000$ used in the prior experiments. Looking at the ratios between the accesses done by MSan and by QMSan, in the vast majority of cases QMSan has to instrument more operations. We suspect the outliers `json` and `woff2` may reflect unwanted effects from the MSan instrumentation restraining the optimization opportunities for the clang compiler, especially for the write case of `woff2`. Remarkably, the overwhelming amount of store operations in the program code of `openssl` explains the higher run-time overheads that we experience when fuzzing it. This analysis brought to our attention another design opportunity: we believe these writes occur in pathological loops where the compiler can apply hoisting if the target and the initialization status cannot change. In future work, we would like to study optimizations for these behaviors in the context of the opportunistic detector, for which it would suffice to mark only once a given address as initialized in an execution (as long as that memory remains allocated).

VI. OTHER RELATED WORKS

Binary Sanitization: Recently, a few works have explored binary sanitization in combination with fuzzing. As we mentioned in Section I, none of them has tackled UUM errors.

RetroWrite [11] proposes an approach based on reassembleable disassembly to instrument binary software. RetroWrite uses static analysis to insert both fuzzing instrumentation and sanitization machinery to detect for addressability issues. The work shows how static techniques can produce good results in binary analysis, outperforming QEMU-based dynamic analysis by 4.5x. After the publication of RetroWrite, the developers of AFL++ have significantly optimized the fork server and other runtime features for use with the User Emulation of QEMU [28], largely reducing the performance gap between dynamic and static binary instrumentation. As a limitation,

	MSan			QMSan			Ratio		
Subject	Load	Store	Total	Load	Store	Total	Load	Store	Total
c-ares	10 250	3 977	14 227	13 603	9 659	23 262	1.33	2.43	1.64
guetzli	41 623 113	143 232 234	184 855 347	71 042 102	136 017 694	207 059 796	1.71	0.95	1.12
json	26 492 307	15 586 824	42 079 131	25 112 824	19 801 050	44 913 874	0.95	1.27	1.07
libxml2	144 661 204	39 678 510	184 339 714	188 860 261	137 939 926	326 800 187	1.31	3.48	1.77
openssl	23 660 580	5 702 123	29 362 703	64 886 395	98 870 336	163 756 731	2.74	17.34	5.58
pcre2	21 959 153	14 434 450	36 393 603	51 990 182	34 862 467	86 852 649	2.37	2.42	2.39
re2	38 172 849	16 540 906	54 713 755	52 042 314	41 401 158	93 443 472	1.36	2.50	1.71
woff2	236 241 446	176 314 501	412 555 947	320 190 558	105 619 783	425 810 341	1.36	0.60	1.03
						geomean	1.55	2.24	1.73

TABLE VIII
FTS subjects: Number of shadow memory accesses from program code done in MSan and QMSan.

RetroWrite can only work with position independent code, as its design necessitates to be able to use relocation information. Most recently, ARMore [13] has brought the capabilities of RetroWrite to the AArch64 domain, removing the position independent code requirement and adding other enhancements.

QASan [12] shows how sanitization can be implemented with dynamic binary translation to detect addressability issues while performing fuzz testing. Being both implemented inside the TCG component of the User Emulation of QEMU, our work shares similarities with the design of QASan, although the instrumentation required to detect addressability issue is conceptually very different (e.g., load and store instructions suffice). Addressability issues can only be detected on heap buffers, while stack variables are unsupported. This is due to the fact that ASan-like instrumentation requires redzones to be placed in between allocations, in order to detect possible overflows when accessing memory. While this is easy to achieve by interposing on heap management functions and redirect the addresses they work on, stack-allocated storage cannot be moved around to make room for redzones.

MTSan [21] uses Memory Tagging Extension (MTE), a hardware feature available in the AArch64 architecture, to perform binary sanitization for fuzzing. With this feature, MTSan can detect errors between stack and global objects without the need to insert redzones. Unfortunately, this solution still suffers from the lack of known boundaries between allocations in both stack and global objects, so it can have false positives. To tackle this problem, MTSan implements a technique called progressive object recovery, which uses information about the multiplicity of fuzzing runs to probabilistically reconstruct the layout of stack and global objects, thus minimizing the number of false positives.

Optimizing Sanitizers: Other works have explored techniques to reduce the performance cost of address sanitization for both compiler-level and binary-level implementations.

FuZZan [19] argues that the root cause of sanitization overhead when fuzzing is heavyweight metadata (namely, the shadow memory), which is designed to support frequent metadata operations over long executions. To reduce this overhead, FuZZan proposes a new lightweight metadata structure that trades startup and teardown costs for slightly higher per-access costs; then, it also proposes a technique to switch between metadata structures at run-time to automatically detect which is the most effective, remembering this decision throughout the fuzzing campaign to reduce the slowdown globally.

SanRazor [18] focuses on reducing the number of sanitization checks. It first gets coverage information by doing a profiling phase; then, it uses this information to analyze sanitization checks and determines which can be removed without losing precision. In particular, it uses both static and dynamic information to detect which checks are dominated by others, thus removing the need for the dominated one.

Asan-- [29] reduces the overhead from sanitizer checks by introducing new optimizations. It first detects and removes unsatisfiable checks by implementing a new lightweight static approach based on control flow traversal and basic constant propagation. Then, it removes recurring checks by using dominator analysis. Further, it tries to merge neighbor checks to reduce the cost of multiple checks. Finally, it optimizes checks in loops by detecting and merging them. By doing so, it reduces run-time overhead by 41.7% on SPEC CPU 2006.

Optimizations similar in spirit to those from SanRazor and Asan-- may be most effective in challenging cases like the `openssl` one we encountered in our evaluation, and provide helpful benefits to most programs in general. As we mentioned in Section III-E and later in Section V-D, we look forward to exploring this angle soon.

VII. Conclusion

We presented QMSan, the first practical, fuzzer-friendly, and performant solution for UUM sanitization when fuzzing. Through an opportunistic, multi-layer design, we trade a residual number of test case executions under full instrumentation (i.e., program and libraries, with shadow propagation) for a remarkably faster operation during fuzzing, reusing knowledge about presumed violations that were false positives and instrumenting only a program's load and store operations. This solves the practical limitation of MSan, avoiding false positives in exchange for a residual risk of false negatives, which are preferable for practical value (just as it happens with ASan and libraries). We showcase the practical value of QMSan by exposing 44 new UUM bugs in 13 programs. Our hope is that, pending positive feedback from the community, QMSan will be considered for inclusion in OSS-Fuzz.

Acknowledgements

We thank our anonymous reviewers for their feedback. We are grateful to Andrea Fioraldi and Luca Di Bartolomeo for the help they provided at different stages of the project. This work was supported by the Italian MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU through projects SERICS (PE00000014) and Rome Technopole (ECS00000024), by ERC H2020 under grant 850868, and by SNSF PCEGP2_186974.

References

[1] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "SoK: Sanitizing for security," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1275–1295.

[2] B. Garmany, M. Stoffel, R. Gawlik, and T. Holz, "Static detection of uninitialized stack variables in binary code," in *Computer Security – ESORICS 2019*, K. Sako, S. Schneider, and P. Y. A. Ryan, Eds. Cham: Springer International Publishing, 2019, pp. 68–87.

[3] M. Corporation, "Cve-2010-0263," 2010. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-0263

[4] ——, "Cve-2015-1763," 2015. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1763

[5] ——, "Cve-2023-50188," 2023. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-50188

[6] ——, "Cve-2024-32878," 2024. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2024-32878

[7] M. Payer, "The fuzzing hype-train: How random testing triggers thousands of crashes," *IEEE Security & Privacy*, vol. 17, no. 1, pp. 78–82, 2019.

[8] E. Stepanov and K. Serebryany, "MemorySanitizer: fast detector of uninitialized memory use in C++," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2015, pp. 46–55.

[9] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *2012 USENIX annual technical conference (USENIX ATC 12)*, 2012, pp. 309–318.

[10] jonathanmetzman, "Projects with MSAN disabled because of upgrade to Ubuntu 20.04," 2021. [Online]. Available: https://github.com/google/oss-fuzz/issues/6294

[11] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1497–1511.

[12] A. Fioraldi, D. C. D'Elia, and L. Querzoni, "Fuzzing binaries for memory safety errors with QASan," in *2020 IEEE Secure Development (SecDev)*. IEEE, 2020, pp. 23–30.

[13] L. Di Bartolomeo, H. Moghaddas, and M. Payer, "ARMore: Pushing love back into binaries," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 6311–6328.

[14] R. Malmain, A. Fioraldi, and F. Aurélien, "LibAFL QEMU: A library for fuzzing-oriented emulation," in *BAR 2024, Workshop on Binary Analysis Research, colocated with NDSS 2024*, 2024.

[15] J. Seward and N. Nethercote, "Using Valgrind to detect undefined value errors with bit-precision." in *USENIX Annual Technical Conference, General Track*, 2005, pp. 17–30.

[16] D. Bruening and Q. Zhao, "Practical memory checking with Dr. Memory," in *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE, 2011, pp. 213–223.

[17] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: data race detection in practice," in *Proceedings of the workshop on binary instrumentation and applications*, 2009, pp. 62–71.

[18] 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)*, 2021, pp. 479–494.

[19] Y. Jeon, W. Han, N. Burow, and M. Payer, "FuZZan: Efficient sanitizer metadata design for fuzzing," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 249–263.

[20] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida, "ParmeSan: Sanitizer-guided greybox fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2289–2306.

[21] X. Chen, Y. Shi, Z. Jiang, Y. Li, R. Wang, H. Duan, H. Wang, and C. Zhang, "MTSan: A feasible and practical memory sanitizer for fuzzing COTS binaries," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 841–858.

[22] P. Borrello, A. Fioraldi, D. C. D'Elia, D. Balzarotti, L. Querzoni, and C. Giuffrida, "Predictive context-sensitive fuzzing," in *Network and Distributed System Security Symposium (NDSS)*, 2024.

[23] D. Bruening, Q. Zhao, and S. Amarasinghe, "Transparent dynamic instrumentation," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE '12)*. ACM, 2012, pp. 133–144.

[24] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, 2018, pp. 2123–2138.

[25] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks, "Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1683–1700.

[26] J. Jung, S. Tong, H. Hu, J. Lim, Y. Jin, and T. Kim, "Winnie: Fuzzing windows applications with harness synthesis and fast cloning," in *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS 2021)*, 2021.

[27] J. Ba, G. J. Duck, and A. Roychoudhury, "Efficient greybox fuzzing to detect memory errors," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3561161

[28] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[29] Y. Zhang, C. Pang, G. Portokalidis, N. Triandopoulos, and J. Xu, "Debloating address sanitizer," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 4345–4363.

[30] "How to use the persistent mode in AFL++'s QEMU mode," https://github.com/AFLplusplus/AFLplusplus/blob/stable/qemu_mode/README.persistent.md, accessed: 2024-10-15.

Appendix A
Additional Accuracy Experiments

To further compare QMSan's accuracy with other state-of-the-art binary-level solutions, as anticipated in Section V-C, we generate $100,000$ test cases from saturated queues and execute them with individual rounds in Valgrind, Dr. Memory, and both our accurate and opportunistic detectors. We conduct this experiment on the closed-source software of Section V-B and extend the analysis to the 8 FTS subjects of Section V-A.

We first check whether our accurate detector and the two state-of-the-art tools yield consistent answers, so as to spot potential false negatives and false positives. We start by checking if every test case that raises a UUM violation in one of the three systems does so also in the other two, obtaining consistent results[8]. This finding confirms that our accurate detector can report a buggy test case to the fuzzer (and the user) like existing tools would do, further validating our implementation. We then attempt automatic deduplication of all the reports to group UUM errors with identical stacktrace.

Table IX shows the results of this experiment. QMSan's accurate detector detects the same UUM errors (both in

[8]We experienced false positives with Dr. Memory with both versions of `nconvert`. One single false positive occurred in all executions due to imperfect handling of string manipulation operations where safe operations happen on potentially uninitialized data. By inspecting the log from Dr. Memory, we could confirm the tool is still able to detect, after those false positives, all the (true) UUM errors as in Valgrind and our accurate detector when a test case raises one. For this reason, in Table IX we report Dr. Memory can detect the same unique UUM violations as the other two systems.

Program	Unique reported violations			
Name	Opportunistic	Accurate	Valgrind	Dr. Memory
cuobjdump 12.3	2	2	2	2
cuobjdump 12.4	0	0	0	0
nconvert 7.136	9	20	20	20
nconvert 7.155	10	17	17	17
nvdisasm 12.3	0	0	0	0
nvdisasm 12.4	0	0	0	0
pngout	28	28	28	28
rar 6.11	1	1	1	1
rar 7.0	0	0	0	0
c-ares	0	0	0	0
guetzli	0	0	0	0
json	0	0	0	0
libxml2	16	16	16	16
openssl	1	1	1	1
pcre2	1	1	1	1
re2	2	2	2	2
woff2	0	0	0	0

TABLE IX
ADDITIONAL ACCURACY EXPERIMENTS USING RECORDED TEST CASES. THE TWO SETS OF PROGRAMS ARE, RESPECTIVELY, THE PROPRIETARY PROGRAMS FROM RQ1 AND THE FTS SUBJECTS FROM RQ3.

quantity and identity) as existing tools. This result comes to no surprise as its accurate detector is designed to have identical capabilities to the state of the art. The attentive reader would notice that, for the proprietary programs, the counts are higher than those from Table III, which instead are bugs manually deduplicated with the assistance of developers. For the two versions of `nconvert`, all these test cases here collectively reproduce the same, respectively, 5 and 4 bugs from the RQ2 evaluation. For `pngout`, in the responsible disclosure of the 2 bugs from RQ2, we reported 11 distinct UUM violations and accepted the developer's analysis of 10 being just one logic bug with the argument that they were all about the auto-detection logic of a single file format (`TGA`). This rationale seems applicable also to the 28 crashes found in this experiment[9].

We then simulate a fuzzing session by executing the opportunistic detector with an initially empty map of violations (Section IV) over the $100,000$ recorded test cases. Our goal is to assess whether, in a fuzzing session, the opportunistic detector can expose the same UUM errors as the accurate one. We identify unique violations among those that the opportunistic detector sees as new and supplies to the accurate detector for validation, and that in turn the latter reports as true UUM errors. We provide these violation counts in Table IX.

For most benchmarks, these counts coincide with those from the accurate detector and the other tools. The discrepancies on the two `nconvert` versions are only apparent and explainable with the different meaning of violation in the opportunistic detector. As we explained throughout Section III, the opportunistic detector sees a potential violation when a load operation involves data marked as uninitialized in the shadow memory, whereas the accurate detector (like existing tools) performs shadow propagation and raises a violation at the instruction that uses the uninitialized data. As we explain below, there are no false negatives in these experiments.

By manual analysis of logs, we validate that each unique violation the existing tools found in their analyses directly stems from one uninitialized read that the opportunistic detector identifies (and sees confirmation for) in this experiment. For `nconvert`, we identify distinct violations that are different uses of the same data that a single load instruction reads from memory, which the opportunistic detector reliably detects. In other words, we are dealing with different manifestations (uses) of the same issue (reading an uninitialized data item), and reporting the issue once is sufficient to identify the bug[10].

APPENDIX B
PERSISTENT FUZZING

Binary fuzzing literature typically centers around the use of a fork server for testing, stopping execution when about to enter the `main` function and cloning the process at each run to execute a test case in a clean state. This choice improves fuzzing efficiency by eliminating several costs attributable to operating system, linker, and C library activity. When a program performs time-consuming initialization steps in its code, fuzzing users can also use an optimization, called *deferred initialization*, that moves forward to a manually selected program location the point where the fork server suspends execution. These optimizations are standard in the fuzzing practice and require no changes to the program under test.

Persistent fuzzing is an optimized execution mode where, instead of forking a new process for each test case, the fuzzer reuses one process for multiple test cases. To enable persistent fuzzing, the program (or its portion) under test should operate like a function that one can call multiple times and that resets program state by itself when execution ends: each program run must have no impact on future ones and leave no resource leaks after its completion. For stability reasons, the fuzzer periodically replaces the process with a newly forked one.

While persistent fuzzing can enable important performance gains, it typically requires modifications to the program under test, especially if the program is not a library. For example, many OSS-Fuzz subjects come with harnesses featuring a loop to initialize the required context, invoke the code, and eventually clean up the program state at each execution.

QMSAN's design is naturally compatible with persistent fuzzing. Assuming the program comes with a harness or does not need one, our handling of heap deallocations and function returns automatically restores the status of the shadow memory for heap and stack memory in use to test case execution.

From an implementation perspective, though, binary-level fuzzer architects assume that the program under test is an off-the-shelf binary. For example, while AFL++'s compiler-based instrumentation provides macros to annotate harness code provided for persistent fuzzing, its QEMU-based counterpart only supports restoring a saved state for emulator registers and

[9]We did not follow up on it due to the stated intention of the developer not to fix issues that come from inputs with apparently random contents.

[10]This dynamic loosely reminds us a notable difference between bugs and crashes, as distinct crashes may be different manifestations of the same bug.

pages when execution reaches a specific instruction. That is, the latter has no natural provisions to accommodate a harness for persistent mode, as it does not expect that one can exist.

In preliminary experiments, we tested some FTS subjects with QMSAN by writing a generic harness that reads the input from a file and passes it to the program-specific `LLVMFuzzerTestOneInput()` function, and then setting the emulator to restore registers once our harness regains control. In 1-hour tests, we obtained 2.5-5x speedups consistent with AFL++'s QEMU-mode documentation [30]. This forced interaction, however, may be suboptimal for performance and implementation tweaks may improve these speedups. For example, one may disable state restoration, place our generic harness in a cycle, and extend the QEMU backend to inform AFL++ about test case execution completion through signals. This may also ease the realization of another known fuzzing optimization that replaces input files with a shared memory. As these implementation refinements are conceptually orthogonal to our design, but potentially also of independent interest (e.g., for cross-architecture testing), we leave them to future work.