Fuzzing Binaries for Memory Safety Errors with QASan

Andrea Fioraldi
Sapienza University of Rome
andreafioraldi@gmail.com

Daniele Cono D'Elia
Sapienza University of Rome
delia@diag.uniroma1.it

Leonardo Querzoni
Sapienza University of Rome
querzoni@diag.uniroma1.it

***Abstract*—Fuzz testing techniques are becoming pervasive for their ever-improving ability to generate crashing trial cases for programs. Memory safety violations however can lead to silent corruptions and errors, and a fuzzer may recognize them only in the presence of sanitization machinery. For closed-source software combining sanitization with fuzzing incurs practical obstacles that we try to tackle with an architecture-independent proposal called QASan for detecting heap memory violations. In our tests QASan is competitive with standalone sanitizers and adds a moderate 1.61x average slowdown to the AFL++ fuzzer while enabling it to reveal more heap-related bugs.**

1. Introduction

Memory safety is one of the most desirable properties for software programs: violating it can bring difficult-to-analyze errors, inconsistent results, silent memory corruptions, and ultimately security vulnerabilities [1], [2]. Language designers and runtime architects have come up over the years with different approaches, including automatic safety provers [3], conservative garbage collectors [4], transformation systems for safe dialects [5], and ultimately runtime checks for languages that run in managed environments like Java [6].

Memory unsafe languages like C and C++ are however indispensable for many tasks, such as system programming and performance-sensitive scenarios. Building techniques and tools to identify safety violations in programs before their use in production today is still a heartfelt necessity.

From a security standpoint, memory corruption exploits are getting more sophisticated over time: an attacker may leverage safety violations to carry out control flow hijacking, privilege escalation, and information leakage [7]. While mainstream system defenses such as Address Space Layout Randomization raised the bar for attackers, their reactive nature cannot prevent such attacks in the general case.

Software developers today can benefit from multi-faceted solutions for program testing, with publicly available frameworks for static techniques such as symbolic execution [8], [9], abstract interpretation [10], and bounded model checking [11], and for dynamic approaches such as fuzzing [12] and sanitization [7]. In particular the last few years have seen fuzz testing techniques getting the spotlight due to their ability of efficiently generating crashing test cases for programs [13]. However not all memory safety violations lead to an immediate crash [14], as for instance with accesses to padding bytes inserted for alignment purposes.

Best practices nowadays often combine fuzzing with sanitization [15]. Sanitization tools, or colloquially *sanitizers*, can directly observe incorrect behavior for specific classes of violations as it happens. Sanitizers usually operate by augmenting the program representation to insert tripwires that expose violations of predefined policies.

Instrumenting the source code for sanitization usually brings limited performance overhead, and developers can naturally compose it with fuzzing techniques. This combination is unfortunately precluded to closed-source libraries and programs, which prevail in the software landscape [15].

Sanitization tools available for binary programs typically build on dynamic binary translation frameworks [16]. Two practical factors hinder their composition with binary fuzzing solutions: their standalone nature and the high overheads from the underlying translation technology.

Recently, researchers have prototyped static instrumentation solutions that modify binaries to insert the probes as if they were added during compilation [15]. While rewriting systems have significantly improved, they remain incomplete as they rely on structural properties of the code. For instance, 32-bit platforms are presently out of their reach, while today they are still relevant for e.g. IoT devices [7].

Our Proposal. In this paper we address the practical gap between binary fuzzing and sanitization by proposing a sanitization design that naturally complements existing fuzzing solutions. Its QASan implementation can mimic the capabilities in detecting heap memory violations of ASan (Address Sanitizer [17]), which according to recent research [7] is by far the most widely adopted sanitizer today.

To facilitate its integration in existing fuzzing proposals, QASan builds on top of the QEMU translation framework. In the design we handle memory-related metadata in a way that makes it compatible with heterogeneous architectures, supporting binary rehosting scenarios and reducing the memory pressure on the analyzed target application.

The paper describes a two-pronged evaluation. In the first part we measure the accuracy of QASan in detecting heap violations using a well-known test suite. We then show the end-to-end utility of our proposal by evaluating its integration in the popular AFL++ fuzzer [18]. On a selection of well-tested programs, QASan reveals non-crashing bugs that would be missed by standard fuzzers. When integrated in the normal working of a fuzzer, we measure a performance overhead of 1.26-2.74x for our sanitization scheme, which can be appealing in dynamic testing scenarios.

To foster further research in the area, we share QASan as open source at https://github.com/andreafioraldi/qasan.

2. Background

In this section we describe the main elements behind sanitizers, focusing on heap safety violations. We then introduce instrumentation solutions available for the design of sanitization and fuzzing tools that operate on binary code.

2.1. Memory Sanitization

A recent work [7] describes sanitizers as dynamic bug finding tools that produce precise results valid for the observed execution instance. Unlike exploit mitigations deployed in production by operating systems, the goal of sanitizers is finding errors before a software gets released. For this reason, sanitizers are allowed a bigger resource budget to precisely locate a bug (instead of only detecting a generic error later in time), and possible false alerts can be tolerated as long as their number is manageable [7].

Sanitizers are available for different categories of bugs, such as memory safety violations, type errors, and undefined behavior. A common trait of many solutions is to instrument the code of the program and/or the surrounding software stack with explicit detection sequences that expose violations of a policy, which otherwise could go unnoticed or later cause difficult-to-diagnose crashes or incorrect results [1].

The focus of this paper are memory safety violations occurring in the heap regions, which as we mentioned earlier are widespread in real-world software and can pave the way to several classes of exploitation attacks. A *safety violation* happens when a pointer references an area other than the one belonging to the object that the pointer was initially defined for. Such violations can be spatial or temporal.

Spatial safety violations happen when an access is (partially) outside the bounds of the intended object for the pointer. Allocation defines such bounds: any following arithmetic shall keep a pointer inside one object of interest.

There are two main approaches to detecting spatial violations [1]. *Object-based approaches* track allocated memory objects and check whether memory dereferences fall entirely within a single object. Boundaries of adjacent objects can be distinguished either through range lookup operations, which however can incur very high overheads [1], or by altering the layout of objects to enforce protection mechanisms, trading increased memory usage for better performance. For instance, some sanitizers pad objects with *redzones* of bytes that get marked as invalid in an internal representation, known as *shadow memory*, that they use to validate accesses.

Pointer-based approaches track base and bounds for each pointer, and are more complete than object-based solutions as they can catch for instance dereferences of pointers that point to memory from an object other than their intended one. However they are generally incompatible with uninstrumented code [7] and incur higher overheads as they need to propagate metadata through pointer manipulations.

Temporal safety violations occur instead when a memory dereference happens through a pointer that is no longer valid, i.e. the pointed object is no longer the same as when the pointer was created [19]. Dangling pointers are a common case of temporal violations, as they lead to bugs and also to security vulnerabilities in use-after-free scenarios [7].

Our proposal extends the popular ASan sanitizer to work over binaries and boost fuzzing scenarios. ASan follows an object-based approach for detecting spatial violations. It uses compile-time instrumentation to capture memory operations for their validation, and interposes on memory allocations to update its shadow memory and to pad objects with redzones. It also supports sound, but incomplete detection of temporal violations by delaying memory reallocation operations. We will further detail its internals throughout §3.2 and §3.3.

2.2. Instrumentation for Binary Analysis

The design of instrumentation frameworks for supporting program analyses over binary-only software is a largely studied problem in the programming language, systems, and security research communities [16]. A common requirement for sanitizers, also shared with the fuzz testing scenario that we target for QASan, is the support for *probe insertion* into the program. Probes monitor and mediate classes of instructions that contribute to the object of the analysis, such as memory accesses when detecting memory violations or control flow transfers when tracking path exploration.

To this end binary rewriting techniques have been studied since the early '90s [20]. Frameworks such as DynInst [21] provide static instrumentation using trampolines that overwrite the instructions of interest to invoke analysis code. While this research area keeps evolving [22], supporting real-world COTS binaries has historically proved elusive for a number of factors, such as accurate disassembly without symbol information, ambiguity on indirect-branch targets, use of shared libraries, and dynamic code generation.

Other works explored reassembly [23] to turn the hard-coded code and data references from the output of disassemblers into labels, heavily simplifying subsequent code manipulations and recompilation. Recently RetroWrite [15] uses reassembly to add sanitization probes to x64 position-independent code. Albeit very efficient, these approaches are not general as they depend on characteristics of the compiled code and the platform, with only few settings supported.

A different approach is to build analyses around dynamic binary translation (DBT) systems, which can monitor and potentially alter every instruction as the program is about to execute it [24]. DBT solutions can cover all the cases mentioned earlier as difficult for static techniques, albeit at the price of slower execution, and expose to a (non-adversarial) program the same addresses and data that it would see in a native execution [16]. Among popular frameworks, Pin [25] and DynamoRIO [26] work by manipulating copies of the original instructions, while Valgrind [27] and QEMU [28] translate them to a compilable intermediate representation.

Valgrind is the basis of the popular memcheck memory error detector [27], while DynamoRIO is behind the Dr. Memory tool [29]. With the notable exception of WinAFL [30], neither framework became popular for fuzzing. On the other hand, QEMU has become the technology of choice for a large body of binary fuzzing research

(e.g., [31]–[33]). Some of the reasons behind its popularity are the support for many platforms with a unified design, the simplicity of its Tiny Code Generator component when inserting instrumentation of different kinds, and the performance improvements seen for its User Emulation mode.

3. QASan

This section presents our design, with an initial overview followed by discussions of the main components of QASan (Figure 1) and by an examination of its integration with QEMU-based fuzzers, its limitations, and future directions.

3.1. Overview

QASan follows the object-based approach of ASan to enact its sanitization capabilities over heap accesses made in binary programs. It builds around QEMU User Emulation to intercept read and write operations in the code under analysis, represented by the executable sections of a program and/or libraries of interest. QASan then performs interposition in the address space of the target program for events involving the allocation and release of heap memory.

The design comprises two main components. The first is a QEMU extension that maintains a shadow memory for the target's heap within the address space of the emulator, and augments the Tiny Code Generator (TCG) to insert probes for memory accesses. When the program executes an instrumented memory access, we execute an analysis code inlined in the compiled emulated code to validate the operation or detect an error. Finally, this component also maintains a shadow stack to provide contextual information for allocation sites later involved in memory violations.

The second component is a runtime library preloaded when starting the target program. Its main role is to intercept heap allocation operations, padding objects with redzones for underflow and overflow accesses. It also enforces a quarantine for free operations to detect temporal heap violations.

Once execution starts, the runtime interacts with the QEMU extension through hypercalls to update the shadow representation upon heap memory allocation or release. It also uses hypercalls for faster validation when the target calls commodity library functions for memory manipulation.

3.2. QASan Extension for QEMU

The primary tasks of the QEMU component of QASan are instrumenting memory accesses and hosting the shadow memory representation necessary for their validation.

Memory Accesses. The TCG is a pivotal element in QEMU as it transforms instructions from the target binary architecture to RISC-like emulated TCG operations, and in turn compiles them to instructions that execute natively on the host architecture that runs QEMU.

Intercepting memory accesses at TCG level is rather convenient. QEMU lifts heterogeneous primitives from different architectures using the same load and store TCG operations. Instrumentation completeness easily follows, as

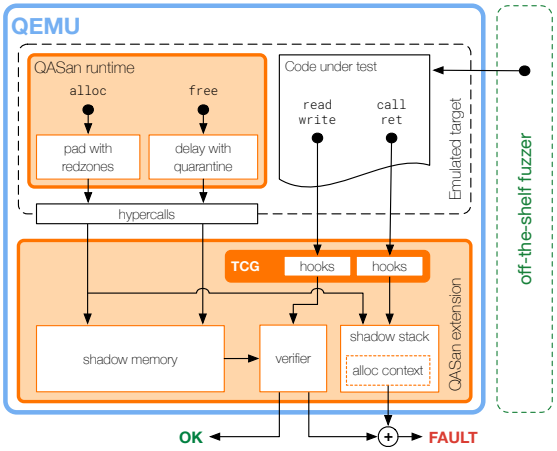

Figure 1. Architecture of QASan and its main components.

SIMD instructions and unbounded `rep`-like patterns are translated with repetitions of such operations.

```
void tcg_gen_qemu_ld_i64(TCGv_i64 val, TCGv addr,
                         TCGArg idx, TCGMemOp memop) {
  [...]
  gen_ldst_i64(INDEX_op_qemu_ld_i64, val, addr, memop, idx);
  switch (memop & MO_SIZE) {
      case MO_64: qasan_gen_load8(addr, idx); break;
      case MO_32: qasan_gen_load4(addr, idx); break;
      case MO_16: qasan_gen_load2(addr, idx); break;
      case MO_8:  qasan_gen_load1(addr, idx); break;
      default:    qasan_gen_load8(addr, idx);
  }
}
```

The excerpt above shows how we instrument a memory load from the TCG engine. After the instruction generation, we add a switch construct to determine the operand size and invoke the QASan helper for that size. The helper verifies if the instruction belongs to a code module to be sanitized and instruments it for validity checking accordingly. Handling memory stores is analogous and omitted for brevity.

Shadow Memory. The design of ASan relies on a shadow memory to deem the validity of a memory read or write access. Instead of mirroring memory with a shadow representation of the same size, ASan uses only a $1/n$ fraction of it building on the observation that heap allocations return addresses aligned to 2^n bytes, with $n \geq 3$ typically.

In particular, in the default setting of $n = 3$ ASan uses 1 byte to store which bytes in a 8-byte memory region are addressable: all of them, the first $k \in \{1..7\}$, or none. ASan hosts the shadow memory in the address space of the program, which is as big as max bytes, by reserving a partition of size $max/8$ at address $offset$. Whenever instrumentation needs to validate an access to an address $addr$, it can efficiently retrieve the corresponding entry in the shadow memory by accessing address $offset + (addr \gg 3)$.

In QASan we chose to allocate the shadow memory region inside the emulator. This approach brings two advantages: we reduce the memory pressure on the analyzed program, which is a known problem for memory-intensive applications and 32-bit architectures [7], and we can use the same indexing scheme for architectures with different addressing modes (e.g., x86 and MIPS) as in the TCG we operate on addresses normalized for the host platform.

To store and check validity information we follow ASan. A zero entry indicates a fully valid 8-byte region, and when fewer a positive integer equals the number of valid first bytes. As unaligned unsafe accesses are very rare [17], for 8-byte load and store operations we only check if the shadow memory entry for the address is zero. For shorter sizes we use a bitmask, for instance for a 1-byte load we check:

```
uintptr_t h = (uintptr_t)addr;
int8_t*   shadow_addr = (int8_t*)(h >> 3) + SHADOW_OFFSET;
int8_t    k = *shadow_addr;
return k != 0 && (intptr_t)((h & 7) + 1) > k;
```

The access is valid on a zero return value: this holds if `k` is zero or larger than the last three bits of the address [17].

Additional Tasks. The QEMU component of QASan ships two further facilities that assist the runtime in the target.

The first one is a *hypercall* mechanism that lets the runtime transfer control to analysis code running in the emulator. As we will explain in the next section, this mechanism is essential to update the shadow memory when the program allocates or releases memory. Similarly to most DBT schemes, the design of QEMU establishes a barrier between the regions accessible to the emulated target and those for emulator and analysis code. This barrier can however be broken for instance for system call translation, when the emulator executes such a call on behalf of the target.

We thus introduce a fictional system call number to trigger a hypercall for QASan. This mechanism is architecture-agnostic and allows the runtime to yield control and transfer data to the analysis code, for instance regarding an allocation operation. We then devise a more efficient, architecture-specific variant that triggers the QASan analysis code directly, avoiding the switch operation on the number in the QEMU system call handler. Namely, we introduce a custom opcode in the instruction set that the TCG lifter for the architecture (e.g., x86) recognizes and translates efficiently.

The second facility is a *shadow stack* to track the calling context (i.e., the sequence of functions currently active on the stack [34]) of heap allocations. This information is crucial to track the origin of heap blocks that later get involved in memory violations. While ASan determines the calling context by unwinding the stack on an allocation event, this solution would not work in our scenario as binaries are often compiled without stack frame pointers used for unwinding. We thus track `call` and `ret` instructions to maintain a shadow stack that mirrors the program's call stack.

3.3. QASan Runtime for Target Execution

The runtime component operates alongside the target program to interpose on heap manipulations for updating the shadow memory and enforcing quarantine policies for detecting temporal violations, and on commodity memory manipulation functions for faster and accurate validation. To implement the runtime we use a dynamically linked library that we preload at program startup to hook function symbols.

Memory Allocation and Release. Similarly to ASan, we hook and replace with specialized implementations functions like `malloc`, `free`, `realloc`, `posix_memalign`, and others used for heap allocation. Interposition is necessary for two tasks: keeping the shadow memory up to date, and padding each buffer with surrounding redzones to detect underflow and overflow errors when dereferencing pointers.

For a newly allocated object we issue a single hypercall to update the shadow memory representation and to mark its surrounding redzones as inaccessible. The hypercall specifies metadata such as address and size for the object, while the size of redzones is a user-configurable global parameter.

One difference with ASan is that it stores allocation metadata (size, thread id, context) within redzones, thus their minimum size is 32 bytes [17]. In QASan we maintain this information—retrieving the calling context from the shadow stack—in the emulator using an interval tree. This choice enables the user to choose smaller redzones when they seek better performance at the cost of possibly more false negatives, while from a design perspective minimizing the intrusiveness in the target memory may turn out useful for extending QASan to whole-system sanitization.

Finally, QASan may detect use-after-free temporal violations like ASan by using a quarantine strategy to delay immediate reallocation of buffers freed by the program. We poison the freed area in the shadow memory so that if the application dereferences a dangling pointer that falls in the region (not yet reallocated) of a previously freed object, the hooks for memory accesses will detect the violation.

Commodity Functions. Programs typically make use of standard library functions to perform routine tasks for memory byte (e.g., `memcpy`) and string (e.g., `strcmp`, `atoi`) manipulation. ASan uses *interceptors* for several such functions from `libc` to validate the involved memory regions before executing them. In QASan we follow a similar strategy, as we can see in interposition excerpt below for `atoi`:

```
int atoi(const char *str) {
  size_t len = __libqasan_strlen(str) + 1;
  QASAN_LOAD(str, len);
  return __lq_libc_atoi(str);
}
```

The symbol hooking mechanism of the runtime rewires program calls to the standard `atoi` function to our specialized version, which makes a single `QASAN_LOAD` hypercall to validate the involved memory buffer efficiently at once before invoking the original `libc` implementation.

When looking for subtle bugs, the user may wish to sanitize memory accesses made in other functions of the library. In ASan this usually implies compiling and linking a sanitized `libc` version for the program. The design of QASan allows for using a standard uninstrumented `libc`, and makes special provisions for its load widening uses.

To explain *load widening* we borrow from a popular discussion on the LLVM mailing list [35]: consider a `char a[22]` array and an operation `a[16]+a[21]`. The LLVM compiler may emit a 8-byte load from `&a[16]` to read both bytes at once when the array is 16-byte aligned and located on stack, yet this load results out of bounds by 2 bytes. The discussion eventually led to introducing optimization metadata that ASan uses to ignore such cases, while binary-only approaches like RetroWrite and QAsan can only detect them as false positives [36]. However when found in stan-

dard library code false alerts of this kind could be a show-stopper for the analysis causing an early termination. We found many load widening instances in `libc` from SIMD instructions defined as inline assembly in its source code[1].

In QASan we opted to rewire a selection of commodity memory manipulation functions, for which in our experience we encountered load widening cases in their implementation (§A), to semantically equivalent, memory-safe counterparts. We then use hot patching to intercept internal calls to such "troublesome" functions from other functions of the `libc` library: since these calls would not be caught by symbol hooking, we insert a trampoline in the entry block of the called functions to forward them to our safe variants.

3.4. Discussion

Fuzzer Integration. The design of QASan can naturally complement existing fuzzing solutions based on QEMU User Emulation. We implemented our QEMU extension in the 3.11 release of the emulator, adding ∼500 LOCs for handling hypercalls and maintaining a shadow stack, and ∼200 for hooking memory accesses and generating validation code in the TCG emulator. Transferring our patches to the AFL++ [18] fuzzing framework required only very few, superficial changes. We believe extending other QEMU-based fuzzers would be similarly simple: our extension does not interfere with the normal working of TCG that they extend with instrumentation, nor alters the code layout and transfers that drive coverage-guided fuzzing [12]. The startup scripts from a testing harness need then a modification to set an environment variable to preload our runtime.

Strengths and Limitations. QASan brings off-the-shelf heap memory sanitization for binary fuzzing, which as we experimentally explore in §4.2 can reveal bugs otherwise likely to be missed. We build it around QEMU due to the latter's practical relevance, however the instrumentation features we strictly need (i.e., memory hooks and system call forwarding) are available in nearly any DBT system [16]. The design is compatible with different architectures, opening the door to rehosting scenarios for, e.g., firmware testing.

Compared to ASan, our main limitation is the inability to check access validity for global variables, stack objects, and intra-object fields. These are well-known pitfalls of binary approaches [7], [15]: such data units are difficult to locate in the first place due to information loss from compilation, then their layout cannot be altered with redzones without patching all the references to them in the code.

The insertion of hooks to validate memory accesses may be suboptimal in terms of performance. This happens for accesses that do not involve the heap or that a compiler can optimize. For the first case we whitelist stack-specific instructions like `pop` in lifters, while a static analysis could exclude more instructions whose access target is statically determinable. The second case applies mainly to unoptimized binary code, for instance when the compiler did not hoist a loop-invariant memory read outside a loop.

Future Directions. A promising avenue for further research is to extend our implementation to full-system sanitization. Fuzzing kernel code is an active research area [37], [38], but sanitizers have traditionally been out of reach for kernel code [7]. Two Google projects [39], [40] can now sanitize 64-bit Linux kernels using source instrumentation, while Windows and 32-bit architectures are presently out of reach.

Another direction could be the integration of MSan [41], a sanitization technique for uninitialized memory reads. With source-based instrumentation ASan and MSan cannot be used together as each would interfere with accesses made to the shadow representation of the other, while keeping such representations in the emulator as in QASan would avoid it.

4. Evaluation

This section describes a two-pronged evaluation. We first review the ability of QASan to find violations in a collection of benchmarks for which the ground truth is known. We then evaluate its end-to-end utility, showing it can expose bugs otherwise missed by a state-of-the-art fuzzer, and measure the overhead added to the fuzzing process by sanitization.

4.1. Memory Violations

We evaluate the memory violation finding abilities of our implementation using the Juliet C/C++ test suite v1.3, which contains a large collection of test cases for over 100 classes of vulnerabilities. Among those we select those pertaining to memory safety violations, namely: `CWE-121` (Stack-based Buffer Overflow), `CWE-122` (Heap-based Buffer Overflow), `CWE-124` (Buffer Underwrite), `CWE-126` (Buffer Over-read), `CWE-127` (Buffer Under-read), `CWE-415` (Double Free), `CWE-416` (Use After Free), `CWE-590` (Free Memory not on the Heap). We leave out test cases that depend on external sources or seek specific values from random oracles, as they inherently demand for static analyses.

Table 1 reports figures collected when compiling the tests for an x64 Linux target. We arranged the categories so that the violations in the first group may involve global storage, heap, or stack memory, while in the second group they are heap-only. Half of the test cases in each group are by design true negatives (TN), i.e., they are memory-safe.

We compare QASan against source-based instrumentation with ASan and a DBT-based approach with the popular Valgrind memcheck. Note that both tools cover more sources of errors than QASan, such as spatial safety violations for stack and global storage in ASan (§3.4), and memory leaks and reading uninitialized memory in Valgrind. To allow for a fair comparison, we disable the latter detections in Valgrind as they are not spatial and temporal violations (§2.1, [7]), while for ASan we use the default configuration.

Interestingly, none of the three tools incur false positives (FP) in our tests, thus the sum of true positives (TP) and false negatives (FN) will account for 50% of the tests, or less when the budget of 3 seconds expires for some tests. In the heap-only group highlighted in bold, QASan is as accurate as

1. ASan handles inline assembly sequences only partially, implying that memory access instrumentation may not be complete for some functions.

TABLE 1. RESULTS FOR THE JULIET TEST SUITE. COLUMNS TP AND FN ARE IN %. TN AND FP WERE ALWAYS 50% AND 0%, RESPECTIVELY.

Category	Tests	QASan		ASan		memcheck	
		TP	FN	TP	FN	TP	FN
CWE-121	5720	21.75	27.55	49.86	0.14	25.82	23.48
CWE-124	1856	39.28	10.72	50.0	0.0	39.28	10.72
CWE-126	1260	22.86	27.14	47.46	2.54	22.86	27.14
CWE-127	1856	25.86	24.14	50.0	0.0	25.86	24.14
CWE-122	6780	47.88	2.12	47.17	2.83	47.88	2.12
CWE-415	1636	50.0	0.0	50.0	0.0	50.0	0.0
CWE-416	786	50.0	0.0	50.0	0.0	50.0	0.0
CWE-590	4560	49.98	0.0	50.0	0.0	50.0	0.0
Total	25030	38,75	11,08	49,05	0,95	39,71	10,13
2nd group	13954	48,95	1,05	48,6	1,4	48,95	1,05

TABLE 2. REPORTED BUGS AND THROUGHPUT FOR AFL++ .

Program	Reported bugs		Executions per second (avg)		
	standard	QASan	standard	QASan	overhead
c-ares	0	1	859	618	1.39x
guetzli	1	1	642	426	1.51x
json	1	2	662	472	1.40x
libxml2	0	2	441	350	1.26x
openssl	0	1	118	43	2.74x
pcre2	16	29	613	457	1.34x
re2	0	0	653	448	1.46x
woff2	0	0	550	246	2.24x

memcheck, and they both outperform ASan as they explore standard library code not modeled by ASan. When we disable `libc` instrumentation, the TP percentage drops to 45.04% for CWE-122 as QASan currently implements less models than ASan, while it stays unaltered for the other 7 categories. For the mixed group of the first four table entries, in three of them QASan and memcheck detect essentially the same errors, i.e., those that involve heap objects. For CWE-121, which comprises only stack violations, memcheck has special provisions to detect accesses that overrun the top of the stack and do not lead to an immediate crash. For the time being we opted not to add extra instrumentation (and thus overhead when fuzzing) for handling such cases. On the contrary, the possibility for ASan to alter the stack layout at the source level brings much higher TP percentages.

4.2. Sanitized Fuzzing

As anticipated in §3.4, we integrated QASan in AFL++, a greybox fuzzing framework that implements state-of-the-art techniques like input-to-state correspondence [42] and single-byte compare coverage [43]. For its QEMU backend, AFL++ uses an efficient forking mechanism to reduce the overhead of DBT recompilations across different executions.

To explore the end-to-end utility of our proposal, we test a subset of 8 programs from the Google Fuzzer Test Suite (FTS) [44] reported in the first column of Table 2. The programs feature memory corruptions and other subtle bugs and are commonly used in the literature: they derive from real-world software tested by the OSS-Fuzz initiative [45], thus are not expected to contain shallow bugs. We make runs of 12 hours on an Intel i7-8565U machine with low background activity, using FTS seeds when available[2].

The number of bugs found when QASan is active is higher than with the standard configuration of AFL++. We verified that all the additional bugs would only be revealed with sanitization enabled, with the exception of a segmentation fault for json missed by AFL++ due to fuzzing entropy. For pcre2 we found many bugs not listed in the documentation of the suite. Among all benchmarks, QASan identified 4 (heap) read overflow, 3 write overflow, 4 read underflow, and 11 use-after-free violations (details in §B).

As for time overheads, we should consider that while DBT approaches are expensive on a single execution, in a well-engineered fuzzer code-cache sharing upon forking amortizes a large fraction of the translation overhead. We thus compare how many executions per second AFL++ completes: with QASan enabled we observe overheads in the range 1.26-2.74x, with a geometric mean of 1.61x.

5. Related Work

This section details related research not covered in §2. Developers have explored custom allocators to detect violations in binaries without instrumenting memory accesses. Guard pages are used in e.g. [47]–[49] to allocate an individual page for each object—to be placed at the end of the page—followed by one unaccessible page to detect overflows. These solutions are incomplete and typically probabilistic, have a high memory footprint, miss underflows in read operations, and may violate alignment rules.

BaseSAFE [50] is a specialized system for testing cellular basebands that combines a drop-in allocator with heap canaries checked with the heavyweight hooks of the Unicorn emulation engine. While its design is tightly coupled to a custom software stack and to partial rehosting of a memory dump, BaseSAFE brings prompt advances to the fuzz testing practice for low-level parsers in embedded targets.

HQEMU [51] reduces the overhead of QEMU using additional threads to reoptimize TCG traces with online profiling information and the heavy-duty LLVM compiler. DBILL [52] later extends it to insert sanitization machinery. While efficient on long-running executions, this approach seems not appealing for fuzzing: a fuzzer attempts a very high number of executions, often with shared code caches, and available CPU cores can be used for concurrent fuzzing.

Song et al. provide an excellent and up-to-date overview of sanitization research in [7]. The interplay of fuzzing and memory violations is studied in ParmeSan [53], which uses sanitization tripwires to steer a fuzzer and expose violations earlier, and in UAFuzz [54], which extends grey-box fuzzing with direct metrics to look for use-after-free bugs.

6. Concluding Remarks

QASan brings a timely addition to the fuzzing realm for exposing memory safety errors in binaries. While its design opens the door to directions like whole-system sanitization, its implementation is mature enough to handle complex software like gcc and LibreOffice with no false positives when invoking `libc` code. We share QASan as open source.

2. For libxml we borrow seeds from the fuzzdata project [46], while for pcre we use an empty seed and provide AFL++ with a dictionary.

References

[1] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for C," in *Proc. of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. ACM, 2009, pp. 245–258. [Online]. Available: https://doi.org/10.1145/1542476.1542504

[2] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 48–62.

[3] J. Berdine, B. Cook, and S. Ishtiaq, "Slayer: Memory safety for systems-level code," in *Computer Aided Verification*. Springer Berlin Heidelberg, 2011, pp. 178–183.

[4] H.-J. Boehm, "Space efficient conservative garbage collection," in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, ser. PLDI '93. Association for Computing Machinery, 1993, pp. 197–206. [Online]. Available: https://doi.org/10.1145/155090.155109

[5] G. C. Necula, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy code," *SIGPLAN Not.*, vol. 37, no. 1, pp. 128–139, Jan. 2002. [Online]. Available: https://doi.org/10.1145/565816.503286

[6] A. Lochbihler, "Making the java memory model safe," *ACM Trans. Program. Lang. Syst.*, vol. 35, no. 4, Jan. 2014. [Online]. Available: https://doi.org/10.1145/2518191

[7] 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)*, 2019, pp. 1275–1295.

[8] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, 2018.

[9] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Conf. on Operating Systems Design and Implementation*, ser. OSDI'08. USENIX Association, 2008, pp. 209–224.

[10] D. Distefano, M. Fähndrich, F. Logozzo, and P. W. O'Hearn, "Scaling static analyses at facebook," *Commun. ACM*, vol. 62, no. 8, pp. 62–70, Jul. 2019. [Online]. Available: https://doi.org/10.1145/3338112

[11] D. Kroening and M. Tautschnig, "CBMC – C bounded model checker," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2014, pp. 389–391.

[12] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "The Fuzzing Book," https://www.fuzzingbook.org/, 2019, [Online; accessed 25-May-2020].

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

[14] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices," in *NDSS 2018, Network and Distributed Systems Security Symposium, 18-21 February 2018, San Diego, CA, USA*, 02 2018. [Online]. Available: http://www.eurecom.fr/publication/5417

[15] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization," 2020.

[16] D. C. D'Elia, E. Coppa, S. Nicchi, F. Palmaro, and L. Cavallaro, "SoK: Using dynamic binary instrumentation for security (and how you may get caught red handed)," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, ser. Asia CCS '19. Association for Computing Machinery, 2019, pp. 15–27. [Online]. Available: https://doi.org/10.1145/3321705.3329819

[17] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-sanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. USENIX Association, 2012, p. 28.

[18] 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)*. USENIX Association, 2020.

[19] M. Payer, *Software Security: Principles, Policies, and Protection*, 0th ed. HexHive Books, April 2019. [Online]. Available: http://nebelwelt.net/SS3P/

[20] A. Srivastava and A. Eustace, "Atom: A system for building customized program analysis tools," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, ser. PLDI '94. ACM, 1994, pp. 196–205. [Online]. Available: http://doi.acm.org/10.1145/178243.178260

[21] B. Buck and J. K. Hollingsworth, "An api for runtime code patching," *Int. J. High Perform. Comput. Appl.*, vol. 14, no. 4, pp. 317–329, Nov. 2000. [Online]. Available: http://dx.doi.org/10.1177/109434200001400404

[22] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar, "A platform for secure static binary instrumentation," in *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '14. ACM, 2014, pp. 129–140. [Online]. Available: http://doi.acm.org/10.1145/2576195.2576208

[23] S. Wang, P. Wang, and D. Wu, "UROBOROS: Instrumenting stripped binaries with static reassembling," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 236–247.

[24] P. Feiner, A. D. Brown, and A. Goel, "Comprehensive kernel instrumentation via dynamic binary translation," *SIGPLAN Not.*, vol. 47, no. 4, pp. 135–146, Mar. 2012. [Online]. Available: https://doi.org/10.1145/2248487.2150992

[25] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. ACM, 2005, pp. 190–200. [Online]. Available: http://doi.acm.org/10.1145/1065010.1065034

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

[27] N. Nethercote and J. Seward, "Valgrind: A framework for heavy-weight dynamic binary instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. ACM, 2007, pp. 89–100.

[28] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. USENIX Association, 2005, p. 41.

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

[30] Google Project Zero, "WinAFL," 2020. [Online]. Available: https://github.com/googleprojectzero/winafl

[31] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, "Firm-afl: high-throughput greybox fuzzing of iot firmware via augmented process emulation," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1099–1114.

[32] V. Pham, M. Boehme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, 2019.

[33] A. Fioraldi, D. C. D'Elia, and E. Coppa, "WEIZZ: Automatic grey-box fuzzing for structured binary formats," in *Proc. of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3395363.3397372

[34] D. C. D'Elia, C. Demetrescu, and I. Finocchi, "Mining hot calling contexts in small space," *Software: Practice and Experience*, vol. 46, no. 8, pp. 1131–1152, Aug. 2016. [Online]. Available: https://doi.org/10.1002/spe.2348

[35] LLVMdev Mailing List, "Load widening conflicts with Address-Sanitizer," https://lists.llvm.org/pipermail/llvm-dev/2011-December/046322.html, 2011, [Online; accessed 25-May-2020].

[36] RetroWrite project, "Issue #6: Load widening," https://github.com/HexHive/retrowrite/issues/6, 2019, [Online; accessed 25-May-2020].

[37] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels," in *USENIX Security Symposium*, 2017.

[38] H. Liang, Y. Chen, Z. Xie, and Z. Liang, "X-afl: A kernel fuzzer combining passive and active fuzzing," in *Proc. of the 13th European Workshop on Systems Security*, ser. EuroSec '20. ACM, 2020, pp. 13–18. [Online]. Available: https://doi.org/10.1145/3380786.3391400

[39] Google, "KernelAddressSanitizer (KASAN)," 2020. [Online]. Available: https://github.com/google/kasan

[40] ——, "KMSAN (Kernel Memory Sanitizer)," 2020. [Online]. Available: https://github.com/google/kmsan

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

[42] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: fuzzing with input-to-state correspondence," in *26th Annual Network and Distributed System Security Symposium, NDSS*, 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/

[43] M. Jurczyk, "CompareCoverage," https://github.com/googleprojectzero/CompareCoverage/, 2020, [Online; accessed 25-May-2020].

[44] Google, "Set of tests for fuzzing engines," https://github.com/google/fuzzer-test-suite, 2020, [Online; accessed 20-May-2020].

[45] ——, "OSS-Fuzz: continuous fuzzing of open source software," https://github.com/google/oss-fuzz, 2020, [Online; accessed 20-May-2020].

[46] Mozilla Security, "fuzzdata: Fuzzing resources for feeding various fuzzers with input," https://github.com/MozillaSecurity/fuzzdata, 2020, [Online; accessed 20-May-2020].

[47] "libdislocator," https://github.com/mirrorer/afl/tree/master/libdislocator, 2020, [Online; accessed 20-May-2020].

[48] "GWP-ASAN," http://llvm.org/docs/GwpAsan.html, 2020, [Online; accessed 20-May-2020].

[49] B. Perens, "Electric fence malloc debugger," 1993.

[50] D. Maier, L. Seidel, and S. Park, "BaseSAFE: BasebandSAnitized Fuzzing through Emulation," in *13th ACM Conference on Securityand Privacy in Wireless and Mobile Networks (WiSec '20)*, Jul. 2020.

[51] D.-Y. Hong, C.-C. Hsu, P.-C. Yew, J.-J. Wu, W.-C. Hsu, P. Liu, C.-M. Wang, and Y.-C. Chung, "Hqemu: A multi-threaded and retargetable dynamic binary translator on multicores," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12. ACM, 2012, pp. 104–113. [Online]. Available: https://doi.org/10.1145/2259016.2259030

[52] Y.-H. Lyu, D.-Y. Hong, T.-Y. Wu, J.-J. Wu, W.-C. Hsu, P. Liu, and P.-C. Yew, "Dbill: An efficient and retargetable dynamic binary instrumentation framework using llvm backend," in *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '14. Association for Computing Machinery, 2014, pp. 141–152. [Online]. Available: https://doi.org/10.1145/2576195.2576213

[53] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida, "ParmeSan: Sanitizer-guided Greybox Fuzzing," in *USENIX Security*, Aug. 2020.

[54] M.-D. Nguyen, S. Bardin, R. Bonichon, R. Groz, and M. Lemerre, "Binary-level directed fuzzing for use-after-free vulnerabilities," *ArXiv*, vol. abs/2002.10751, 2020.

Appendix A: Load Widening from `libc`

In the development of QASan we encountered several `libc` functions that make use of load widening for performance purposes. This led us to the implementation of slow variants that we use upon interposition on program calls (§3.3). The list of semantically equivalent functions that we ship in the runtime currently comprises 25 functions:

bzero, explicit_bzero, bcmp, memcmp, memmove, memcpy, mempcpy, memchr, memrchr, memmem, strchr, strrchr, strcasecmp, strncasecmp, strcat, strcmp, strncmp, strcpy, strncpy, stpcpy, strdup, strlen, strnlen, strstr, strcasestr

In our experiments this selection was sufficient to avoid false positives when instrumenting the `libc` implementation shipped with the Ubuntu Linux 18.04 releases.

Appendix B: Additional Experimental Findings

In the following we provide additional details on the experiments described in §4.2. We report the version of the FTS benchmarks used in our tests, and the heap safety violations that QASan found when fuzzing them:

Program	Version	Reported violations
c-ares	51fbb47	1 write overflow
guetzli	9afd0bb	-
json	b04543e	-
libxml2	v2.9.2	1 read overflow, 1 write overflow
openssl	1.0.1f	1 read overflow
pcre2	183 (SVN)	11 use-after-free, 2 read overflow, 1 write overflow, 4 read underflow
re2	499ef7e	-
woff2	9476664	-

To distinguish crashes we do not use the value reported by AFL-derived fuzzers based on code coverage, but we use the instruction pointer value registered for the offending memory violation. Below we report a screenshot of a violation report from QASan when fuzzing libxml2: