Effectively Sanitizing Embedded Operating Systems

Jianzhong Liu[†], Yuheng Shen[†], Yiru Xu[†], Hao Sun[‡], Heyuan Shi[§], Yu Jiang[†*]
[†]KLISS, BNRist, School of Software, Tsinghua University, Beijing, China
[‡]Department of Computer Science, ETH Zurich, Zurich, Switzerland
[§]School of Electronic Information, Central South University, Changsha, Hunan, China

ABSTRACT

Embedded operating systems, considering their widespread use in security-critical applications, are not effectively tested with sanitizers to effectively root out bugs. Sanitizers provide a means to detect bugs that are not visible directly through exceptional or erroneous behaviors, thus uncovering more potent bugs during testing.

In this paper, we propose EmbSan, an embedded systems sanitizer for a diverse range of embedded operating system firmware through the use of dynamic instrumentation of sanitizer facilities and de-coupled on-host runtime libraries. This allows us to perform sanitation for multiple embedded OSs during fuzzing, such as many Embedded Linux-based firmware, various FreeRTOS firmware, and detect actual bugs within them. We evaluated EmbSan's effectiveness on firmware images based on Embedded Linux, FreeRTOS, LiteOS, and VxWorks. Our results show that EmbSan can detect the same criteria of actual bugs found in the Embedded Linux kernel as reference implementations of KASAN, and exhibits a slowdown of 2.2× to 3.2× and 5.2× to 5.7× for KASAN and KCSAN, respectively, which is on par with established kernel sanitizers. EmbSan and embedded OS fuzzers also found a total of 41 new bugs in Embedded Linux, FreeRTOS, LiteOS and VxWorks.

1 INTRODUCTION

Embedded devices have proliferated in recent years, requiring additional security testing to protect the integrity and safety of industries and end users. Failing such can result in disastrous outcomes, including significant financial losses or endangerment of human lives. For example, Heartbleed [4] is an out-of-bounds read bug in OpenSSL that allows attackers to arbitrarily read memory contents, including encryption keys, passwords, etc., from a victim device. In particular, Heartbleed significantly impacts embedded devices that utilize OpenSSL [6], which are not patched as easily as servers, thus leaving many users susceptible to attacks.

Sanitizers are programming tools that detect certain types of bugs that are otherwise undetected by the end user or the system itself. For instance, buffer overflows that fall within allocated memory stay unnoticed, whereas sanitizers that detect such violations can catch the illegal operation on site. In retrospect, using sanitizers during software testing for the SSL libraries will have allowed vendors to detect Heartbleed prior to shipping [17], thus reducing and preventing Heartbleed from manifesting globally.

Embedded operating systems are designed to run specialized pieces of software on embedded devices, which poses numerous difficulties in proposing sanitizer designs that perform various types of sanitizer operations on a wide range of embedded systems. These include: **1)** the diversity of embedded OS implementations and variants, including memory management, device management, actual implementations of the firmware, etc., such that sanitizers from general-purpose systems or other embedded platforms cannot be adapted to a specific system without decent prior knowledge and implementing major changes; **2)** the vast differences in actual embedded platform design, including hardware capacities and capabilities, therefore hindering particular sanitizer features or facilities from being implemented, such as shadow memory, red zones, etc.; **3)** the state of build systems surrounding the various embedded operating systems and their firmware, further preventing sanitizer callbacks and artifacts to be placed within the firmware, notwithstanding the swath of closed-source embedded firmware that are prevalent in the industry. Therefore, there is little prior work towards sanitizing embedded operating systems firmware.

In reality, fuzzing embedded operating systems, like fuzzing its general-purpose counterpart, is performed under full-system emulation, allowing for an unintrusive approach towards sanitizing the system-under-test. Fortunately, this allows us to tap into the emulated target's state and execution process to extract sensitive events and perform operation validation externally. Consequently, our main objective thus is to: **1)** identify the set of sensitive operations common sanitizers intercept, distill a collection of operational semantics that they abide by during a firmware's execution, and determine the instrumentation points during execution; **2)** analyze the firmware-under-test's firmware or source code to determine the platform features and layout details, thus setting the initial state of the target sanitizers; **3)** adapt the kernel sanitizers routines to the host's userspace, allowing for resource-constrained and more capable systems to run the same feature-set of the sanitizers, further reducing the manual efforts involved.

Using these observations, we propose EmbSan, an embedded operating systems sanitizer intended for a wide range of firmware based on different embedded OSs and platforms. EmbSan's design consists of three components, the *Sanitizer Common Function Distiller*, the *Embedded Platform Configuration Prober*, and the *Common Sanitizer Runtime*. Its workflow consists of a *Pre-testing Probing Phase*, where common characteristics of the intended sanitizers are identified, the platform details of the firmware are probed, and the embedded firmware's initial state is then compiled, and a *Testing Phase*, where the firmware is tested using fuzzing or similar methods, while EmbSan intercepts sensitive instructions as required, and passing the arguments as specified to the specific sanitizer's runtime, thus sanitizing the firmware's behavior and reporting any aberrations to the tester.

To validate the effectiveness of our design, we implemented EmbSan on Embedded Linux, FreeRTOS, LiteOS and VxWorks, with support for multiple architectures, including x86, ARM and MIPS. We verify our design using Embedded Linux's native KASAN implementation as a baseline and test EmbSan against numerous confirmed bugs within 5 years. Our results show that EmbSan reaches

*Yu Jiang is the corresponding author.

all design goals and finds all possible bugs. In addition, we measured the overhead of running EmbSan in comparison to running solely the target system under emulation, as well as running systems with their own sanitizers. EmbSan's overhead ranges from 2.2×-3.2× for KASAN and 5.2×-5.7× for KCSAN, compared to 2.2×-2.7× and 5.4×-6.1× of other sanitizers, demonstrating that EmbSan's performance hit is on par with implementing target-specific sanitizers, albeit without the immense manual efforts. Furthermore, EmbSan assisted kernel fuzzers in finding 41 new bugs in firmware based on Embedded Linux, FreeRTOS, LiteOS and VxWorks, respectively.

Our main contributions in this paper are as follows. First, we identify the challenges regarding developing sanitizers for a multitude of multi-architecture embedded operating systems. Next, we propose EmbSan, a sanitizer capable of being easily adapted to and running on a wide selection of embedded operating systems and architectures using methods devised to overcome the previous challenge. Finally, we evaluate EmbSan's effectiveness and demonstrate that it effectively sanitizes embedded systems and finds previously unknown bugs on the systems tested.

2 BACKGROUND AND RELATED WORK

Embedded operating systems are designed to perform specific tasks for use in applications such as medical robotics, autonomous vehicles, IoT devices, etc. These use cases emphasize system efficiency and real-time performance, thus providing fewer functions on a specific selection of platform architectures.

Fuzz testing (fuzzing) is an automatic bug detection technique and has gained a reputation due to its effectiveness in uncovering bugs [1, 2, 9]. Fuzzing performs testing by feeding the target program with large quantities of generated inputs and observing for exceptional behaviors as oracles for the presence of bugs. *Kernel fuzzers* uses fuzzing to find bugs in operating system kernel code. Syzkaller [16] is a state-of-the-art kernel fuzzer that successfully detected numerous bugs within Linux. It uses KASAN and *kcov* [14] to detect bugs and collect coverage information. Rtkaller [12] extends Syzkaller to test RT-Linux's real time scheduling behaviors. Gustave [5] is a fuzzer that uses a custom QEMU board to host embedded operating systems for testing. Tardis [13] is a fuzzer specifically designed to test embedded operating systems. It proposed an OS-agnostic coverage collection mechanism to collect runtime coverage from the target kernel.

Sanitizers are used to detect a target program's erroneous behaviors by modifying the program by marking protection entities, hooking library interfaces with interceptors, and instrumenting sensitive instructions with verification callbacks. The instrumentation during execution calls the sanitizer runtime to update the state of the program and verify any sensitive instruction executed. Commonly used sanitizers include AddressSanitizer (ASAN) [10], Thread Sanitizer (TSAN) [11], Memory Sanitizer (MSAN) [15] and their corresponding kernel sanitizers, such as Kernel Address Sanitizer (KASAN) [7], Kernel Memory Sanitizer (KMSAN) [3] and Kernel Concurrency Sanitizer (KCSAN) [8].

3 DESIGN

The overall architecture of EmbSan is depicted in Figure 1. As shown in the figure, EmbSan consists of three major components, the *Sanitizer Common Function Distiller* (Section 3.1) for identifying the requirements of the sanitizers used during testing, the *Embedded Platform Configuration Prober* (Section 3.2) that specifies the platform details of the target firmware, and the *Common Sanitizer Runtime* (Section 3.3) that intercepts the relevant sensitive instructions during runtime and performs corresponding sanitizer operations. The workflow consists of a *Pre-testing Probing Phase* (Section 3.4) and a *Testing Phase* (Section 3.5).

3.1 Sanitizer Common Function Distiller

The *Distiller* mainly identifies the interface and behavioral characteristics of the sanitizers used. This is a static process, where the sanitizers' interface header files are first fed into the *Distiller* to produce a list of sanitizer's interception APIs. The sanitizers' specific source code files and the instrumentation pass source files are then parsed with the prior list of APIs, where the interception points and call graph of the interfaces are constructed, external resources are identified, and the logic of the sanitizers' APIs are *distilled*, i.e. be converted into an in-house Domain-Specific Language (DSL).

Afterwards, the *Distiller* combines the API specifications of the multiple sanitizers, if required, into a single specification, using the following rules. First, the resulting set of interception points is taken over a union of the individual sanitizer's set. Then, for each interception point, the interface's arguments are also taken as a union of the individual sanitizer's arguments. For arguments that share target data but are not exactly the same, we take the largest possible union of the data and combine them into one argument, and add the corresponding annotations identifying which source APIs the segments belong to into the specifications.

3.2 Embedded Platform Configuration Prober

The *Prober* mainly detects the platform details of the target firmware, including the instruction set capabilities, platform device memory allocation and initial firmware memory layout. This process is mainly dynamic, requiring a pre-testing *dry run* to identify all characteristics. The *Prober* eventually produces a configuration specification and initial setup routine in the aforementioned DSL. As the available selection of embedded OS firmware is highly diverse, we categorize the firmware into three categories and devise individual strategies to probe their initial configurations.

1) For open-source firmware that supports compile-time sanitizer instrumentation, we use a combination of static and dynamic methods to determine its configuration parameters. During the firmware's compilation process, we enable the sanitizer's instrumentation process, but link it with a dummy sanitizer library, where each API is implemented with a platform-specific trapping instruction, such as *vmcall* on x86_64 platforms. In conjunction, we extract the specific locations of each call and their corresponding call definitions and compile them to the aforementioned DSL as the platform specifications. In addition, the point where the system is ready is also inserted with a trapping instruction. We then perform a dry-run of the firmware, where all actions of the sanitizer prior to the point where the system is ready are intercepted and recorded. The actions are also compiled into the DSL as the initial setup routine.

2) For open-source firmware that lacks compile-time sanitizer instrumentation, we instrument instructions that allow us to identify initialization routines and functions that correspond to the

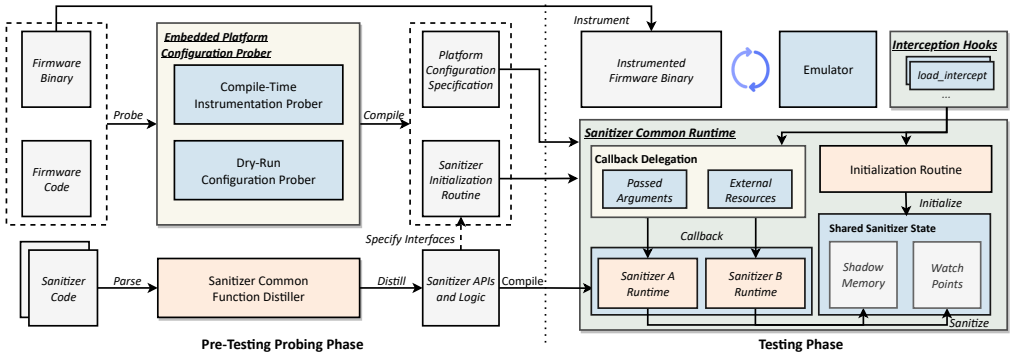

Figure 1: EmbSan's overall component and workflow diagram.

sanitizer's interception functions (such as various *Xalloc()*). The relevant information is compiled into the platform descriptions similar to the aforementioned process. Then, we dry-run the instrumented firmware binary and analyze its behavior up until the firmware's read-to-run state to deduce the initial state of the firmware. Using this information, the *Prober* compiles an initialization routine that sets up the sanitizer's state upon the firmware's initialization. However, this method is not complete, therefore for some platforms and OSs, human intervention may be needed for domain-specific prior knowledge to be added into the descriptions.

3) For closed-source binary-only firmware, we mainly use dynamic methods in conjunction with the tester's prior-knowledge to identify the relevant information. Overall, we use a multi-pass dry-run process with the *Prober* to progressively identify the various artifacts similar to that of the previous categories of firmware, including the point before which is the system's initialization process, function calls with signatures corresponding to the interception function of the sanitizers, and the initial system state and memory layout using probes inserted within the emulator's devices. The *Prober*, with some manual intervention, produces the platform details and initialization routines in the DSL.

As some sanitizer functionalities are only supported through compile-time instrumentation, such as on-stack and static variable *redzones*, i.e. protection zones around memory-allocated objects that allow KASAN to detect off-by-N out-of-bounds accesses, only the first category of firmware is sanitized with support for such functionality. The difference in their effectiveness in bug detection will be discussed in depth in the evaluation.

3.3 Common Sanitizer Runtime

The *Runtime* component mainly accepts the descriptions of the sanitizers and firmware from the probing phase, compiles the specific sanitizer runtimes, then intercepts the firmware's execution process for pre-determined sanitizer-sensitive instructions and interception functions and transfers the execution to each functional sanitizer's runtime library for state update or operation validation.

While there are established techniques for interrupting the emulator's execution process such as Virtual Machine Introspection, we choose to modify the emulator's execution engine itself for better efficiency. To do so, the *Runtime* modifies the emulator's execution engine by inserting callback probes, which are specified by the sanitizers, into the translated code templates. For instance, in QEMU/TCG, one of the emulators that EmbSan supports, for the *load* instruction, which is specified by KASAN and KCSAN, the *Runtime* modifies its translation template by inserting a call to a delegate function *load_intercept()*, where all parameters and resources that are required by the sanitizers, as specified in the DSL, are extracted symbolically, and are then passed to the relevant sanitizer interception functions.

The *Runtime* also maintains the state of all sanitizer functions, such as a unified shadow memory, that records information for multiple sanitizer functionalities. This allows the conservation of memory resources on the host machine and simplifies the complexity involved with transforming descriptions in the DSL into actual sanitizer initialization, maintenance and validation routines.

As firmware with sanitizer instrumentation available allows for direct insertion of callbacks to sanitizer routines, the *Runtime* component thus supports direct hypercalls from the emulator and redirects the calls to the specific sanitizer interfaces, thus improving overhead statistics in such cases.

3.4 Pre-Testing Probing Phase

The initial phase for EmbSan requires the tester to prepare the firmware binary and relevant descriptions as laid out in the aforementioned sections. The specific steps are given below.

First, the tester needs to determine the sanitizers needed, and extract reference implementations from OS kernels, such as Linux's KASAN and KCSAN implementations. The header files containing the API definitions and the actual source code are passed to the *Distiller*, which then produces the descriptions of the interception functions and logic of the sanitizers in the DSL.

The tester will also need to analyze the firmware for available source code, and if so, verify its support for compile-time sanitizer instrumentation. Then, according to the firmware's classification, the tester will invoke either of the three operation modes of the *Prober* and produce the platform specifications and initialization routines, specified using the DSL.

Finally, the firmware needs to be prepared for dynamic testing. For source-code-available embedded OS firmware, this is largely similar to that of the *Prober*'s instrumentation process, where the ready-to-run state is hooked to invoke the initialization routine, and for firmware with sanitizer instrumentation support available,

the instrumentation is enabled with sanitizer linkage to the same dummy sanitizer library described above.

3.5 Testing Phase

After the preparation process, the tester can begin fuzzing or other forms of dynamic testing with EmbSan enabled. As aforementioned, the *Runtime* component accepts the DSL descriptions of the sanitizer interfaces and runtime logic, in addition to the firmware's platform configuration and sanitizer initialization routines. The *Runtime* component then compiles the sanitizer runtime logic and hooks all relevant operations to the emulator's execution process, including sanitizer-relevant instructions and function calls. Testing is then allowed to commence, where the sanitizer will initialize upon the firmware reaching the ready-to-run state, update the runtime's internal state upon the firmware invoking state maintenance operations, and validate any sensitive operations as specified by the original sanitizer.

4 EVALUATION

We perform the following experiments to validate EmbSan's correctness and effectiveness in comparison to established tools, benchmark its ability in conjunction with state-of-the-art kernel fuzzers to discover new bugs and measure the runtime overhead cost of delivering such functionality and its comparison to the relevant sanitizers. To further measure the difference between using compile-time instrumentation and dynamic instrumentation, we run the target kernels under compile-time instrumentation, designated as *EmbSan-C*, and dynamic instrumentation, as *EmbSan-D*.

The firmware we selected for evaluation is determined through the following criteria. First, the firmware should be widely used, thus demonstrating EmbSan's ability to find bugs that potentially affect a wide range of users. Second, the selected firmware should also cover different types of functionalities, such as routers, IoT firmware, etc. Finally, the firmware should be based on different embedded operating systems. The list of tested firmware is in Table 1. Thus, we chose firmware based on OpenWRT, OpenHarmony and others that allow us to cover Embedded Linux, LiteOS, FreeRTOS and VxWorks on x86, ARM and MIPS platforms, utilizing both EmbSan-C and EmbSan-D.

Table 1: List of embedded firmware used in EmbSan's evaluation process, and their respective base operating system, system architecture, instrumentation mode, source code availability, and fuzzer used to test the firmware.

Firmware	Base OS	Architecture	Inst. Mode	Source	Fuzzer
OpenWRT-armvirt	Embedded Linux	ARM	EmbSan-C	Open	Syzkaller
OpenWRT-bcm63xx	Embedded Linux	MIPS	EmbSan-D	Open	Syzkaller
OpenWRT-ipq807x	Embedded Linux	ARM	EmbSan-C	Open	Syzkaller
OpenWRT-mt7629	Embedded Linux	ARM	EmbSan-C	Open	Syzkaller
OpenWRT-rtl839x	Embedded Linux	MIPS	EmbSan-D	Open	Syzkaller
OpenWRT-x86_64	Embedded Linux	x86	EmbSan-C	Open	Syzkaller
OpenHarmony-rk3566	Embedded Linux	ARM	EmbSan-C	Open	Tardis
OpenHarmony-stm32mp1	LiteOS	ARM	EmbSan-D	Open	Tardis
OpenHarmony-stm32f407	LiteOS	MIPS	EmbSan-D	Open	Tardis
InfiniTime	FreeRTOS	ARM	EmbSan-D	Open	Tardis
TP-Link WDR-7660	VxWorks	ARM	EmbSan-D	Closed	Tardis

Our experiments were conducted on a computer with an AMD Ryzen 7 5800X processor, 128GiB of DDR4 memory and running x86_64 Ubuntu Linux 22.04. The compilers used to build the respective kernels are GCC 12.2 and LLVM 14.0. We used the latest versions of Syzkaller and Tardis to test the embedded firmware. We also used QEMU 7.1.0 to run the target firmware on the host system. All quantitative experiments were repeated 10 times, following generally accepted fuzzing evaluation guidelines, in order to reduce statistical errors.

Table 2: Comparison of sanitizing capabilities on previously found bugs between EmbSan-C, EmbSan-D and KASAN.

Bug Type	Kernel Ver.	Location	EmbSan-C	EmbSan-D	KASAN
Out-of-bounds	5.17-rc2	*ringbuf_map_alloc*	Yes	Yes	Yes
Use-after-free	5.19	*ieee80211_scan_rx*	Yes	Yes	Yes
Out-of-bounds	5.17-rc1	*bpf_prog_test_run_xdp*	Yes	Yes	Yes
Use-after-free	5.17	*btrfs_scan_one_device*	Yes	Yes	Yes
Use-after-free	5.19-rc1	*post_one_notification*	Yes	Yes	Yes
Use-after-free	5.19-rc1	*post_watch_notification*	Yes	Yes	Yes
Out-of-bounds	5.17-rc6	*watch_queue_set_filter*	Yes	Yes	Yes
Null-pointer-deref	5.17-rc8	*__free_pages*	Yes	Yes	Yes
Out-of-bounds	5.17	*vxlan_vnifilter_dump_dev*	Yes	Yes	Yes
Out-of-bounds	5.19	*imageblit*	Yes	Yes	Yes
Out-of-bounds	5.19-rc4	*bpf_jit_free*	Yes	Yes	Yes
Use-after-free	5.17-rc6	*null_skcipher_crypt*	Yes	Yes	Yes
Use-after-free	5.18-rc6	*bio_poll*	Yes	Yes	Yes
Use-after-free	5.18	*blk_mq_sched_free_rqs*	Yes	Yes	Yes
Use-after-free	5.18-rc7	*do_sync_mmap_readahead*	Yes	Yes	Yes
Use-after-free	5.18	*filp_close*	Yes	Yes	Yes
Use-after-free	5.17-rc4	*setup_rw_floppy*	Yes	Yes	Yes
Use-after-free	5.18-next	*driver_register*	Yes	Yes	Yes
Use-after-free	5.17-rc4	*dev_uevent*	Yes	Yes	Yes
Out-of-bounds	6.0	*run_unpack*	Yes	Yes	Yes
Use-after-free	5.19	*ath9k_hif_usb_rx_cb*	Yes	Yes	Yes
Use-after-free	5.19-rc1	*vma_adjust*	Yes	Yes	Yes
Use-after-free	6.0-rc7	*nilfs_mdt_destroy*	Yes	Yes	Yes
Out-of-bounds	5.7-rc5	*fbcon_get_font*	Yes	No	Yes
Out-of-bounds	4.17-rc1	*string*	Yes	No	Yes

4.1 Comparison with Native Sanitizers

We first validate EmbSan's soundness in detecting bugs in comparison with Linux's native implementations. As KCSAN raises many false positives, thus reproducing any bugs found is problematic, we perform this on KASAN. For Embedded Linux, we fetched recent bug reports from syzbot's dashboard and extracted those issued by KASAN into a data set containing 25 bugs, all of which are reproducible with reproducer programs. We then compiled the specific kernel versions in each bug report and their respective reproducer programs, and executed the kernel image on QEMU with EmbSan-C, EmbSan-D and KASAN enabled, respectively.

The results of this experiment are shown in Table 2. Other than the last two bugs, all sanitizer implementations are capable of catching each bug during execution. These bugs consist of the following types: slab cache out-of-bounds access, use-after-free, null-pointer-dereferencing, global out-of-bounds accesses, etc. Most of the bugs fall into the first two categories, namely slab cache out-of-bounds and use after free bugs. Realistically, the majority of Embedded Linux bugs discovered fall mostly into these two categories.

We specifically searched for bugs that EmbSan-D theoretically cannot handle, such as global variable out-of-bounds and stack out-of-bounds bugs, to examine the difference in capabilities between EmbSan-C and EmbSan-D. Eventually, we found the last two bugs, where the former dates from 2020 and the latter is from 2018. Both of which are global out-of-bounds bugs. In accordance with our expectations, EmbSan-D was incapable of detecting such violations, as it lacks redzones around global objects, in contrast to EmbSan-C and KASAN that do have such features. We also confirmed that it

was indeed the redzones inserted during compile-time that allowed EmbSan-C to discover such a violation.

Therefore, we demonstrate that EmbSan's effectiveness fully meets our original design goals. In detail, EmbSan-C can detect the same criteria of memory violations as KASAN does, while EmbSan-D, due to a lack of compile-time information, is slightly weaker than the former two. However, as the proportion of use-after-free and slab out-of-bounds bugs are the majority of recently detected bugs, EmbSan-D's weakness is diminished when considering its potential to adapt to a wider range of embedded operating systems than EmbSan-C or KASAN.

4.2 EmbSan's Effectiveness

EmbSan is designed to effectively sanitize embedded operating systems in testing environments. Thus, we deployed both EmbSan-C and EmbSan-D in real-world testing environments. We employ state-of-the-art kernel fuzzers including Syzkaller and Tardis to conduct tests on the aforementioned embedded firmware. Specifically, we use Syzkaller to fuzz Embedded-Linux-based firmware, whereas, for firmware based on LiteOS, FreeRTOS and VxWorks, we have extended Tardis's capabilities with corresponding executor programs and interface specifications.

Table 3: Classification of the 41 new bugs found by EmbSan on various firmware based on Embedded Linux, LiteOS, FreeRTOS and VxWorks.

Firmware	Bug type			
	OOB Access	UAF	Double Free	Race
OpenWRT-armvirt	5		1	
OpenWRT-bcm63xx	3	2		
OpenWRT-ipq807x	3	1	1	
OpenWRT-mt7629	2		2	
OpenWRT-rtl839x	1	1	1	
OpenWRT-x86_64	5			2
OpenHarmony-rk3566	2	1		
OpenHarmony-stm32mp1	1			
OpenHarmony-stm32f407	2			
InfiniTime	2	1		
TP-Link WDR-7660	2			

The fuzzing campaigns were executed over a period of 7 days. All found bugs have been deduplicated and are reproducible. In summary, EmbSan found a total of 41 new bugs in the tested firmware. We show the classification of found bugs in Table 3, and the full list is shown in Table 4. As we can see in the table, EmbSan is capable of assisting testing tools such as fuzzers in detecting new bugs in firmware that span several operating systems and architectures.

For the firmware such as *OpenWRT-x86_64* that have native KASAN or KCSAN support, we also replayed the reproducer programs of the bugs that were found on this firmware using EmbSan with their native KASAN or KCSAN implementations enabled. We found that the bugs can be reproduced by using the native implementations, demonstrating EmbSan's soundness in finding bugs in various embedded firmware.

Therefore, EmbSan can perform sanitizing services for embedded operating system firmware under dynamic testing conditions such as fuzzing and deliver bug finding effectiveness on par with native KASAN and KCSAN implementations for a significantly wider selection of embedded firmware.

Table 4: List of the 41 previously unknown bugs found by EmbSan during kernel fuzzing.

Firmware	Base OS	Arch.	Location	Bug Type
OpenWRT-armvirt	Embedded Linux	ARM	fs/nfs_common	OOB Access
OpenWRT-armvirt	Embedded Linux	ARM	net/netfilter	OOB Access
OpenWRT-armvirt	Embedded Linux	ARM	net/wireless	OOB Access
OpenWRT-armvirt	Embedded Linux	ARM	drivers/net/ethernet/marvell	OOB Access
OpenWRT-armvirt	Embedded Linux	ARM	drivers/net/ethernet/realtek	OOB Access
OpenWRT-armvirt	Embedded Linux	ARM	drivers/net/ethernet/atheros	Double Free
OpenWRT-bcm63xx	Embedded Linux	MIPS	drivers/bluetooth	OOB Access
OpenWRT-bcm63xx	Embedded Linux	MIPS	drivers/dma/bcm2835-dma	OOB Access
OpenWRT-bcm63xx	Embedded Linux	MIPS	drivers/scsi/aic7xxx	OOB Access
OpenWRT-bcm63xx	Embedded Linux	MIPS	fs/btrfs	UAF
OpenWRT-bcm63xx	Embedded Linux	MIPS	drivers/net/wireless/broadcom	UAF
OpenWRT-ipq807x	Embedded Linux	ARM	drivers/net/ethernet/broadcom	OOB Access
OpenWRT-ipq807x	Embedded Linux	ARM	drivers/net/ethernet/broadcom	OOB Access
OpenWRT-ipq807x	Embedded Linux	ARM	net/sched	OOB Access
OpenWRT-ipq807x	Embedded Linux	ARM	drivers/net/wireless/ath	UAF
OpenWRT-ipq807x	Embedded Linux	ARM	fs/fuse	Double Free
OpenWRT-mt7629	Embedded Linux	ARM	drivers/net/ethernet/mediatek	OOB Access
OpenWRT-mt7629	Embedded Linux	ARM	fs/nfs	OOB Access
OpenWRT-mt7629	Embedded Linux	ARM	net/core	Double Free
OpenWRT-mt7629	Embedded Linux	ARM	drivers/dma/mediatek	Double Free
OpenWRT-rtl839x	Embedded Linux	MIPS	drivers/net/ethernet/realtek	OOB Access
OpenWRT-rtl839x	Embedded Linux	MIPS	drivers/net/bluetooth/realtek	UAF
OpenWRT-rtl839x	Embedded Linux	MIPS	fs/netrom	Double Free
OpenWRT-x86_64	Embedded Linux	x86	drivers/iommu	OOB Access
OpenWRT-x86_64	Embedded Linux	x86	drivers/net/ethernet/realtek	OOB Access
OpenWRT-x86_64	Embedded Linux	x86	drivers/net/ethernet/stmicro	OOB Access
OpenWRT-x86_64	Embedded Linux	x86	drivers/net/wireless/intel/iwlwifi	OOB Access
OpenWRT-x86_64	Embedded Linux	x86	drivers/net/wireless/broadcom/b43	OOB Access
OpenWRT-x86_64	Embedded Linux	x86	fs/btrfs	Race
OpenWRT-x86_64	Embedded Linux	x86	fs/btrfs	Race
OpenHarmony-rk3566	Embedded Linux	ARM	fs/nfs	OOB Access
OpenHarmony-rk3566	Embedded Linux	ARM	fs/nfs_common	OOB Access
OpenHarmony-rk3566	Embedded Linux	ARM	net/sched	UAF
OpenHarmony-stm32mp1	LiteOS	ARM	fs/vfs	OOB Access
OpenHarmony-stm32f407	LiteOS	MIPS	fs/vfs	OOB Access
OpenHarmony-stm32f407	LiteOS	MIPS	fs/fat	OOB Access
InfiniTime	FreeRTOS	ARM	src/libs/littlefs/	OOB Access
InfiniTime	FreeRTOS	ARM	src/drivers/Spi	OOB Access
InfiniTime	FreeRTOS	ARM	src/drivers/St7789	UAF
TP-Link WDR-7660	VxWorks	ARM	pppoed	OOB Access
TP-Link WDR-7660	VxWorks	ARM	dhcpsd	OOB Access

4.3 Runtime Overhead

We compare the firmware's execution time with and without EmbSan enabled, and that of a natively-sanitized version, if available. The tasks the firmware runs to measure the overhead are the merged corpus acquired after completing the previous experiment. We evaluate the overhead of EmbSan for KASAN-relevant and KCSAN-relevant functionalities separately to compare with available native sanitizers. The firmware tested is then further divided into the following classifications for comparison: architecture, base operating system, EmbSan instrumentation method used. The overhead evaluation results are shown in Figure 2.

We observe that EmbSan's KASAN functionalities on Embedded Linux-based firmware exhibit a slowdown of 2.2×-2.5× for EmbSan-C, while EmbSan-D achieves a slowdown of 2.7×-2.8×. EmbSan's KCSAN functionalities on the other hand achieve a 5.2×-5.7× slowdown for EmbSan-C. We compare these statistics to Embedded Linux's native sanitizers KASAN and KCSAN, which incur an overhead of 2.2×-2.7× and 5.4×-6.1×, respectively. We further analyzed their runtime composition using tools such as *perf* to inspect the performance logs for the sanitizers running on Embedded Linux, and found that EmbSan requires more instructions to conduct instrumentation and interception calls due to context switches and argument reconstruction, but as native sanitizers run in the guest instance, thus its runtime routines are translated, which is slower than the native execution speed on EmbSan. We conclude that their respective overheads are on par with each other, with

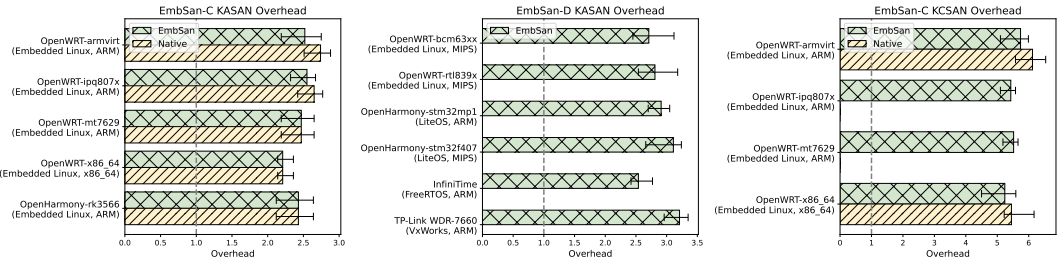

Figure 2: Comparison of runtime overhead statistics between EmbSan and native KASAN & KCSAN sanitizers, and further subdivision into instrumentation modes EmbSan-C and EmbSan-D on the evaluated firmware based on Embedded Linux, LiteOS, FreeRTOS and VxWorks.

EmbSan occasionally performing slightly better than native sanitizers, thus exhibiting an acceptable overhead level for Embedded Linux-based firmware.

For firmware based on LiteOS, FreeRTOS and VxWorks, we observe the slowdown for KASAN functionalities ranges from 2.5×-3.2×, which is similar to that on Embedded Linux, demonstrating EmbSan's efficiency when adapted for different embedded operating systems and platforms.

In summary, we conclude that EmbSan's overhead is well within expectations, with the compile-time instrumented approach achieving even higher than the built-in KASAN, while the dynamic instrumented approach still maintains a reasonable overhead, given its ability to instrument a wider selection of kernels.

5 DISCUSSION

Adaptability of EmbSan: In contrast to adapting existing sanitizers to a new kernel, porting EmbSan is fairly straightforward, which mainly consists of constructing relevant descriptions as described in Section 3. Adapting new sanitizer functionalities to EmbSan is also simple, requiring developers to write runtime code accordingly and designate which instructions to instrument and what interfaces should be called in such invocations.

Bug Detection Potential: The number of new bugs detected by EmbSan in our evaluation is not representative of the EmbSan's full bug detection capabilities. This is due to Syzkaller and Tardis, while being state-of-the-art, still require more system call specifications for embedded firmware. During the evaluation, we found that many modules in VxWorks, LiteOS and FreeRTOS were not covered over the entire fuzzing campaign due to a lack of relevant system call descriptions. Therefore, with a more capable fuzzer, EmbSan has the potential to discover an even greater amount of kernel bugs.

6 CONCLUSION

Our design of EmbSan addresses the difficulties of effectively sanitizing embedded operating systems in testing environments. EmbSan provides a solution for testing environments that can tackle these obstacles. Our evaluation results show that EmbSan is as performant as currently established KASAN and KCSAN implementations, can run as efficient, and is more adaptable to a wide range of kernels and instruction set architectures. EmbSan assisted kernel fuzzers to find 41 new bugs in firmware based on Embedded Linux, LiteOS, FreeRTOS and VxWorks, thus demonstrating its potential in testing a wide selection of embedded kernels with diverse architectures.

7 ACKNOWLEDGEMENTS

This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000) and NSFC Program (No. 92167101, 62021002).

REFERENCES

[1] Peng Chen and Hao Chen. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725, 2018.
[2] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1967–1983, Santa Clara, CA, August 2019. USENIX Association.
[3] The Linux Kernel Developers. The kernel memory sanitizer (kmsan), 2023. https://www.kernel.org/doc/html/next/dev-tools/kmsan.html.
[4] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, page 475–488, New York, NY, USA, 2014. Association for Computing Machinery.
[5] Stéphane Duverger and Anaïs Gantet. Gustave: Fuzz it like it's app. *DMU Cyber Week*, 2021.
[6] Imran Ghafoor, Imran Jattala, Shakeel Durrani, and Ch Muhammad Tahir. Analysis of openssl heartbleed vulnerability for embedded systems. In *17th IEEE International Multi Topic Conference 2014*, pages 314–319, 2014.
[7] Google. Kernel address sanitizer. https://www.kernel.org/doc/html/latest/dev-tools/kasan.html.
[8] Google. Kernel concurrency sanitizer. https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html.
[9] lcamtuf. American fuzzy lop, 2013. https://lcamtuf.coredump.cx/afl/.
[10] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 28, USA, 2012. USENIX Association.
[11] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, page 62–71, New York, NY, USA, 2009. Association for Computing Machinery.
[12] Yuheng Shen, Hao Sun, Yu Jiang, Heyuan Shi, Yixiao Yang, and Wanli Chang. Rtkaller: State-Aware Task Generation for RTOS Fuzzing. *ACM Trans. Embed. Comput. Syst.*, 20(5s), sep 2021.
[13] Yuheng Shen, Yiru Xu, Hao Sun, Jianzhong Liu, Zichen Xu, Aiguo Cui, Heyuan Shi, and Yu Jiang. Tardis: Coverage-guided embedded operating system fuzzing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2022.
[14] SimonKagstrom. Kcov. https://github.com/SimonKagstrom/kcov.
[15] Evgeniy Stepanov and Konstantin Serebryany. Memorysanitizer: fast detector of uninitialized memory use in c++. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 46–55. IEEE, 2015.
[16] Dmitry Vyukov and Andrey Konovalov. Syzkaller: an unsupervised coverage-guided kernel fuzzer, 2015. https://github.com/google/syzkaller.
[17] David A. Wheeler. How to prevent the next heartbleed, Jul 2020. https://dwheeler.com/essays/heartbleed.html.