Lemix: Enabling Testing of Embedded Applications as Linux Applications

Sai Ritvik Tanksalkar
Purdue University

Siddharth Muralee
Purdue University

Srihari Danduri
Purdue University

Paschal Amusuo
Purdue University

Antonio Bianchi
Purdue University

James C Davis
Purdue University

Aravind Kumar Machiry
Purdue University

Abstract

Dynamic analysis, through rehosting, is an important capability for security assessment in embedded systems software. Existing rehosting techniques aim to provide high-fidelity execution by accurately emulating hardware and peripheral interactions. However, these techniques face challenges in adoption due to the increasing number of available peripherals and the complexities involved in designing emulation models for diverse hardware. Additionally, contrary to the prevailing belief that guides existing works, our analysis of reported bugs shows that high-fidelity execution is not required to expose most bugs in embedded software. Our key hypothesis is that security vulnerabilities are more likely to arise at higher abstraction levels.

To substantiate our hypothesis, we introduce Lemix, a framework enabling dynamic analysis of embedded applications by rehosting them as x86 Linux applications decoupled from hardware dependencies. Enabling embedded applications to run natively on Linux facilitates security analysis using available techniques and takes advantage of the powerful hardware available on the Linux platform for higher testing throughput. We develop various techniques to address the challenges involved in converting embedded applications to Linux applications. We evaluated Lemix on 18 real-world embedded applications across four RTOSes and found 21 new bugs, in 12 of the applications and all 4 of the RTOS kernels. We report that Lemix is superior to existing state-of-the-art techniques both in terms of code coverage ($\sim$2X more coverage) and bug detection (18 more bugs).

1 Introduction

Society's dependence on low-powered Microcontroller Unit (MCU) based devices (*e.g.,* IoT devices), has significantly increased, controlling various aspects of our daily lives, including homes [1], transportation [90], traffic management [80], and the distribution of vital resources like food [61] and power [59]. The adoption of these devices has seen rapid and extensive growth, with an estimated count of over 50 billion devices by the end of 2020 [27]. Vulnerabilities in the software controlling these devices have far-reaching consequences [2, 96] due to the pervasive and interconnected nature of these devices, as exemplified by the infamous Mirai botnet [53] and more recent URGENT/11 [91] vulnerabilities. It is important to detect such vulnerabilities proactively. Various works [97] show that dynamic analysis, especially fuzzing [52], is effective at vulnerability detection in web and desktop software. However, the dynamic analysis of embedded systems [28] is challenging [56, 94] because of the close interaction with hardware and the lack of Operating System (OS) abstractions. The lack of robust and readily available dynamic analysis tools (comparable to those for x86 systems) further imposes engineering challenges.

To mitigate this, *rehosting* [22] has emerged as an effective technique. By decoupling firmware from its hardware dependencies and enabling execution within an emulated environment, rehosting facilitates deeper exploration and analysis of embedded software without the constraints of physical hardware. Existing rehosting techniques mainly focus on achieving high-fidelity execution without hardware and focus on modeling peripheral interactions through manually created models [15], pattern-based model generation [23], or models built using machine learning techniques [81, 31]. They depend on the availability of an MCU-specific Instruction Set Architecture (ISA) emulator and require considerable engineering effort [95] to configure different peripherals. *We hypothesize that this high-fidelity execution is not required for vulnerability detection, and a coarse approximation of program behavior is sufficient*. We validate our hypothesis through a preliminary analysis of previously reported bugs (§ 3.2.2). We find that most bugs arise in higher-level software, not in architecture-specific code like inline assembly.

Starting from this observation, in this paper, we present Lemix, a novel approach to rehost embedded applications as Linux applications (for x86), which we call LeApps, with the goal of improving vulnerability detection capability in embedded software with minimal engineering effort. Lemix enables the use of dynamic analysis techniques readily available for Linux applications, such as sanitizers [74] on embedded applications. However, converting embedded applications to x86 Linux applications and enabling dynamic analysis poses challenges, *i.e.,* preserving execution semantics, retargeting to different ISA, and handling peripheral interactions. We maintain execution semantics by leveraging the Linux Portable Layer, which comes as a part of most of the prevalent RTOSes (§ 4.1.1). We use an interactive refactoring approach (§ 4.1.2) to handle ISA retargeting. We tackle peripheral interactions (§ 4.1.4) by first identifying MMIO addresses through constant address analysis and using runtime instrumentation to feed peripheral data through standard input, thereby eliminating the need for precise peripheral models. We also weaken peripheral state-dependent conditions to improve code coverage, which is often limited by these conditions that are difficult for a fuzzer to bypass. To further improve testing, we apply a function-level fuzzing approach based on available research [47, 58] that directly invokes the target function with appropriate arguments generated from the input. Taken together, these design choices form a novel rehosting methodology that enables efficient bug discovery in embedded applications without sacrificing practical effectiveness, as demonstrated by our findings.

We evaluated LEMIX on 18 real-world embedded applications ranging across four RTOSes, including FreeRTOS, Nuttx, Zephyr, and Threadx. These RTOSes support major semiconductor platforms like Qualcomm, NXP, Nordic [73, 85, 99, 64] etc. We show that our approach can successfully convert applications to LEAPP with only a little manual effort. We tested LEAPPS by using whole-program fuzzing and function-level fuzzing and found 21 previously unknown bugs with 14 out of 18 applications effected by these bugs. Our ablation study shows that each of our techniques significantly contribute to the overall effectiveness of LEMIX. Finally, comparative evaluation against the state-of-the-art shows that LEMIX is superior at improving code coverage (∼2X more coverage) and bug detection (18 additional bugs).

In summary, the following are our contributions:

- We propose LEMIX, an extensible framework to rehost embedded applications as x86 Linux applications (*i.e.,* LEAPPS) without emulation or physical devices.
- We design various analysis techniques to tackle challenges in maintaining execution semantics, retargeting, and handling peripheral interactions. We also design techniques to improve the testing and code-coverage of LEAPPS.
- We evaluated LEMIX on 18 embedded applications across four Real Time Operating Systems (RTOSes) and found 21 previously unknown bugs, most of which are confirmed and fixed by the corresponding vendors.
- Our comparative evaluation against state-of-the-art shows that LEMIX is superior in code coverage and bug detection.

RTOS	Low Fidelity	High Fidelity
FreeRTOS	20	2
Zephyr	26	7
RIOT	14	2
Total	**60 (85%)**	**11 (15%)**

Table 1: Our analysis of the CVEs from the Rust4Embedded survey [69], [76] indicates that 60 out of 71 (85%) require low-fidelity execution. Table 8 (Appendix) has category-wise breakdown of the bugs.

2 Background and Threat Model

We provide the necessary background of our target embedded systems (§ 2.1) and information about their software architecture (§ 2.2), along with our threat model (§ 2.3).

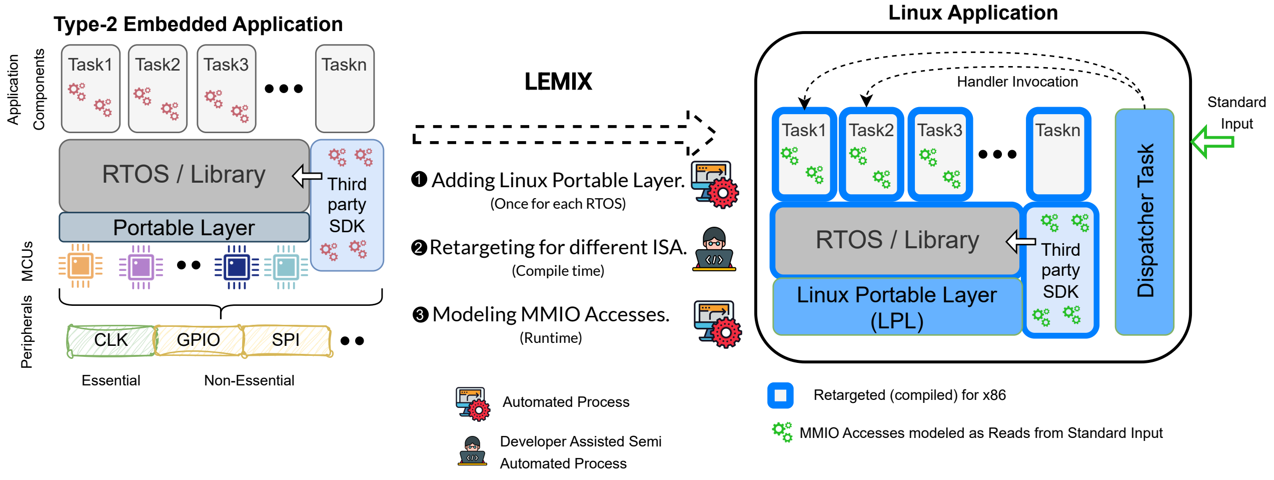
2.1 Type-2 Embedded Systems

Embedded systems perform a designated task with custom-designed software and hardware. Following previous systematization works [56, 22], these systems can be categorized into three types: Type-1 systems use general purpose OSs retrofitted for embedded systems, *e.g.,* Embedded Linux; Type-2 systems use an RTOS, a class of OS that provides timing guarantees, minimal hardware abstraction, and prioritizes tasks to meet strict timing constraints critical for real-time applications, and Type-3 systems use no OS abstractions.

In this work, we focus on Type-2 systems, which consist of an RTOS combined with application code. Type-2 designs are common in safety-critical scenarios, supported by the availability of safety-certified RTOSes [93, 70, 89], which comply with guidelines like those set by MISRA [7] and provide real-time guarantees [84]. As shown in Figure 1, they have a layered design [78] and decouple the application components from the underlying RTOS kernel. Most RTOSes modularize their code base to capture all the hardware-specific functionalities within a portability layer specialized per MCU.

2.2 Portability Layers

As shown in Figure 1, RTOSes depend on a portable architecture to enable easy support for the diverse set of available CPU architectures and boards. Specifically, the portable layer provides header files that define interfaces between the hardware-agnostic kernel and the various MCU-specific ports. The RTOS kernel above the portable layer contains hardware-agnostic code. The hardware-specific implementations, containing interactions with specific MCU registers, memory regions, and peripherals, are contained in MCU ports, which are compiled and linked with the kernel. As a result of this, an embedded application designed for a specific CPU architecture can run on a different CPU architecture by replacing the current MCU port with that of the new architecture [39].



Type-2 Embedded Application
Application Components
Task1
Task2
Task3
Taskn
RTOS / Library
Third party SDK
Portable Layer
MCUs
Peripherals
CLK
GPIO
SPI
Essential
Non-Essential
LEMIX
❶ Adding Linux Portable Layer. (Once for each RTOS)
❷ Retargeting for different ISA. (Compile time)
❸ Modeling MMIO Accesses. (Runtime)
Automated Process
Developer Assisted Semi Automated Process
Linux Application
Handler Invocation
Standard Input
Dispatcher Task
Linux Portable Layer (LPL)
Retargeted (compiled) for x86
MMIO Accesses modeled as Reads from Standard Input

Figure 1: Architecture of a Type-2 Embedded System and overview of LEMIX approach to convert it to a Linux Application.

To improve testability and aid embedded firmware development, many RTOSes also provide ports for various host operating systems such as Linux and Windows. We refer to these ports as the *Native Portable Layer (NPL)* and this includes the *Linux Portable Layer (LPL)* and the *Windows Portable Layer (WPL)*. These native ports allow embedded applications built on these RTOSes to be run on respective desktop operating systems as native applications. Native ports use host-provided implementation to simulate various embedded functionalities. For example, the Linux Portable Layer (LPL) of the FreeRTOS [25] and Zephyr [100] operating systems use Linux *pthreads* to simulate tasks, *signals* to simulate interrupts, and *timers* to simulate clocks in the application. We provide details in Appendix of our extended report [87].

2.3 Threat Model

Embedded applications receive inputs from a variety of sources, such as network interfaces, external storage devices (*e.g.,* SD cards, USB), user-provided inputs via buttons or screens, and peripherals accessed through Memory Mapped I/O (MMIO). In our threat model, we assume that *the attacker can control all inputs to the embedded application, including those coming from peripherals accessed via MMIO accesses*. Specifically, all values through MMIO reads are fully controlled by the attacker. The goal of the attacker is to trigger vulnerabilities in the embedded application.

This threat model is reasonable from the Defense in Depth perspective [57] and has been used in several other works [5, 48, 41]. Also, from a software resilience standpoint, it is important to reasonably validate data received from external entities (such as peripherals) to avoid arbitrary failures. For instance, in Listing 15 (Extended Report [87]), blindly trusting the data from MMIO `GPIOx->LATCH` (Line 2) could result in an infinite loop (Line 23), causing DoS.

3 Motivation

Dynamic analysis, such as fuzzing, is shown to be an effective technique for vulnerability detection [52]. Scalable dynamic analysis of Type-2 embedded applications requires an instrumentation capability (*e.g.,* through an emulator) and hardware independence. One of the most popular approaches is *rehosting* [22], where an unmodified embedded firmware will be executed or rehosted in a virtualized environment. One of the main challenges in rehosting is to achieve execution fidelity. The existing rehosting techniques can be categorized according to the developer/analyst effort and execution fidelity as shown in Figure 2. Ideally, we want to achieve *the highest execution fidelity with the least analyst effort in a hardware-independent manner* — a known hard problem and the holy grail of rehosting [22]. Most of the recent rehosting techniques try to achieve high execution fidelity and mainly focus on automated techniques to precisely model peripheral interactions — which are hard to generalize across peripherals. Furthermore and more importantly, such high-fidelity execution may not be needed to detect most vulnerabilities.

3.1 Execution Fidelity (EF)

Adapting[1] the categorization from Wright *et al.,*[95], Execution Fidelity (EF) in embedded systems can be broadly grouped into four categories:
Language Semantic Fidelity (S): The degree to which the

[1] We build upon the broader categorization of Wright *et al.,* by introducing more granular taxonomies, enabling a more detailed assessment of execution fidelity specifically in the context of embedded systems.

```
1   int32_t tud_msc_read10_cb(uint32_t lba, uint32_t offset,
2   void* buffer, uint32_t bufsize)
3   {
4     // out of ramdisk
5     if ( lba >= DISK_BLOCK_NUM ) {
6       return -1;
7     }
8     /* Attacker can offset to sensitive memory */
9     uint8_t const* addr = msc_disk[lba] + offset;
10    /* Controlled write to known memory
11        may cause undefined behavior */
12    memcpy(buffer, addr, bufsize);
13    return (int32_t) bufsize;
14  }
```

Listing 1: Lack of bounds check on offset in `tud_msc_read10_cb` allows out-of-bounds read from `msc_disk[lba]`, potentially leading to information disclosure or undefined behavior.

execution preserves the high-level language (*e.g.,* C/C++) semantics intended by the programmer, including control flow, data types etc.

Assembly Execution Fidelity (A): The correctness of executing assembly instructions which constitutes instruction-level behavior and any deviations due to instrumentation etc. In contrast to **S**, which focuses on high-level program behavior, **A** pertains to low-level execution behavior as specified by the processor's instruction set architecture (ISA).

Peripheral Handling Fidelity (P): The extent to which peripheral interactions (*e.g.,* memory-mapped I/O) are accurately modeled or handled during execution. While **A** ensures correct instruction behavior, P focuses on the correctness of effects on peripheral device interaction, requiring hardware modeling beyond the instruction level.

Clock Fidelity (C): The accuracy of timing behavior with respect to real-time constraints such as instruction timing, interrupts, system clock behavior etc.

For the sake of simplicity, we define Execution Fidelity (EF) as $< S, A, P, C >$, where each component is categorized as **Low (L)**, **Medium (M)** or **High (H)**. While we adopt this discrete structure for clarity, finer gradations or even a continuous scale (Examples in Appendix § A.0.1) may offer further insights and are left for future work. This definition of EF also provides a way to categorize existing works. For instance, hardware-in-the-loop approaches, such as AVATAR [98], redirect all peripheral handling to the real board and execute the embedded firmware on the emulator. The split execution does not preserve the relative clock semantics between the emulator and actual hardware and only achieves partial clock fidelity, *i.e.,* C = M (Medium). The EF achieved by these approaches can be specified as $< H, H, H, M >$.

3.2 Bug Manifestation Fidelity (BMF)

BMF is the minimum fidelity required to reach and observe the effects of the bugs of interest. BMF varies according to the type of bugs. For instance, to observe scheduling bugs, we need an accurate clock fidelity, *i.e.,* C, in addition to the other components, depending on where the bug is. If scheduling bugs do not involve assembly, we do not need A. We therefore analyze known vulnerabilities in embedded software to understand the BMF required for memory corruption vulnerabilities (a common class of vulnerabilities). Specifically, which execution aspects out of **S, A, P, C** (§ 3.1) are required and which of them can be approximated.

3.2.1 Empirical Data

We manually analyzed 84 publicly reported vulnerabilities in C/C++ software taken from the recent work by Sharma *et al.,* [75] to identify what degree of fidelity is required to manifest them. This included CVEs with available patch information from open-source RTOSes, *i.e.,* FreeRTOS, Zephyr, and RIOT. We considered only the common case of memory corruption vulnerabilities, omitting categories such as weak authentication and SQL injection. Memory corruption vulnerabilities comprised 71 out of the 84 vulnerabilities.

For each CVE, we identified the target vulnerability and affected function by manually analyzing the CVE description and the corresponding patch. We then check if the vulnerability can be triggered with low-fidelity rehosting. Specifically, we target vulnerabilities characterized by an EF of $< H, L, M, M >$, as defined in § 3.1. We consider those that meet all requirements to be triggerable with low-fidelity rehosting, else high fidelity is needed. Listing 8 (Appendix) shows an example of a CVE requiring high-fidelity rehosting and Listing 7 (Appendix) shows an example of a CVE requiring low-fidelity. We summarize our results in Table 1. More details can be found in Table 8 (Appendix). Our analysis is further confirmed by recent work [88], which detected various vulnerabilities through low-fidelity dynamic analysis.

3.2.2 BMF For Embedded System Software

Based on our empirical study § 3.2.1, the BMF required for most of the memory corruption bugs is $< H, L, M, M >$, which is what LEMIX targets. Following the definitions of Wright *et al.,* [95], BMF for most memory corruption vulnerabilities can be approximated to module-level execution fidelity. Specifically, we should be able to execute a module (*i.e.,* a group of functions) with *enough fidelity* to expose a bug. Listing 1 demonstrates a motivating example of a bug we discovered in TinyUSB. The `tud_msc_read10_cb` function lacks bounds checking on the `offset` parameter, allowing out-of-bounds reads from the `msc_disk` array (at line 9), which can cause potential information disclosure or even undefined behavior, depending on how the buffer is further used. We do not need a high-fidelity execution to detect the bug in Listing 1. We just need to execute the function `tud_msc_read10_cb` and pass a large number as `offset`. We also need the capability to detect out-of-bound memory access (at line 9), which is challeng-

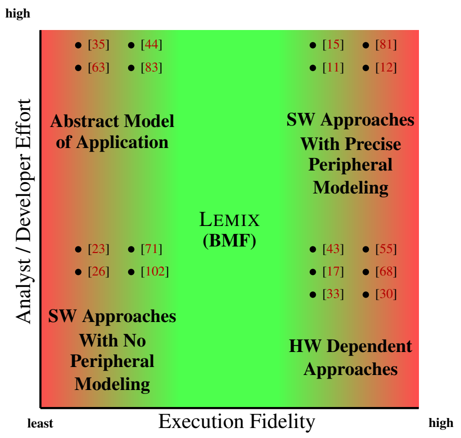


Figure 2: Existing Rehosting Techniques v/s LEMIX

ing in embedded systems because of the lack of memory protection mechanisms [56]. Although we do not require precise peripheral models to trigger the bug, achieving BMF or module-level execution fidelity without them is challenging. As mentioned in § 2.1, embedded applications are organized into a set of tasks and use a real-time scheduler to trigger the tasks. To execute the function `tud_msc_read10_cb` in Listing 1, we need to ensure the task containing the function gets executed, which further depends on the scheduler, which requires precise models for the clock peripheral. *Can we achieve BMF without explicitly providing precise peripheral models?* In summary, we need the capability to execute embedded application, handle MMIO accesses (*i.e.,* provide data on reads and ignore writes), and detect memory safety violations.

3.3 The Idea

Dynamic analysis challenges like execution environment and detectability have been well studied for Linux applications on standard ISAs (e.g., x86, x64), with many effective solutions [18, 79, 9]. Prior work, such as AoT [40], extracts components from complex systems (e.g., Linux kernel) into testable user-space applications. Our goal is to convert embedded applications into Linux applications to enable BMF and make existing dynamic analysis techniques [51] applicable. Srinivasan *et al.,* [82] recently showed this is feasible by manually converting three simple FREERTOS applications. However, designing a generic technique involves tackling the following challenges.

- **(Ch1) Preserving Execution Semantics.** Linux applications, by default, follow single-threaded execution. However, embedded applications (as explained in § 2.1) are engineered in terms of event-driven tasks and are multithreaded [32]. Simply replacing RTOS files with their POSIX equivalents (LPL) often leads to unintended errors during integration. Incorporating a POSIX-compatible RTOS requires a systematic and automated mechanism. This involves more than just file replacements; it necessitates careful adaptation to preserve the embedded system's original task-based and event-driven execution semantics.
- **(Ch2) Retargeting to different ISAs.** Though majorly developed in C, embedded applications use various non-standard and embedded toolchain-specific C features not supported by traditional compilers for desktop ISAs, *e.g.,* x86. The presence of inline ISA-specific assembly (*e.g.,* of ARM) further complicates retargeting (*i.e.,* compiling) for other ISAs. We need to have a mechanism to compile an embedded application for common desktop ISAs.
- **(Ch3) Handling Peripheral Interactions.** Embedded systems directly interact with peripherals, mostly through a dedicated set of MMIO addresses [65]. It is crucial to distinguish these MMIO addresses from regular memory accesses because they correspond to physical hardware components, and improper handling can lead to incorrect behavior.

4 LEMIX

We design LEMIX, an interactive framework enabling effective dynamic analysis of embedded applications by converting them to Linux applications, which we call LEAPP. The novelty of LEMIX lies in recognizing and harnessing the BMF insight, *i.e.,* rehosting with just enough execution fidelity to keep most security bugs triggerable, thereby significantly reducing the manual effort and complexity typically associated with generalizing full system emulation. The right side of Figure 1 shows the summary of our approach to tackling the challenges (§ 3.3) in converting to LEAPP. The LEMIX framework has two phases as illustrated in Figure 3. In Phase 1, we convert the given embedded application into LEAPP using static analysis techniques and compiler instrumentation. We use an interactive approach to tackle certain complex code idioms during retargeting. We also design instrumentation techniques for LEAPPs to improve the effectiveness of dynamic analysis, specifically random testing. In Phase 2, we focus on testing LEAPP. We support two modalities, whole-program, and function-level testing, providing a holistic testing infrastructure.

4.1 Phase 1: Analysis Friendly LEAPP

This phase generates a dynamic analysis-friendly LEAPP from a given embedded application and the target RTOS configuration. This part tackles challenges (1-3) from § 3.3.

4.1.1 Handling execution semantics using LPL (Ch 1)

As explained in § 2.1, embedded applications rely on RTOS functions for their execution semantics. For instance, an application for FREERTOS uses `xTaskCreate` function to create a

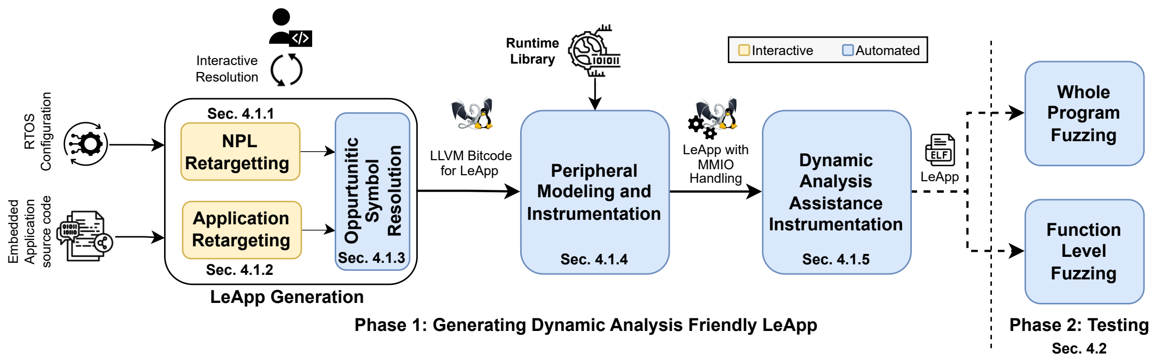


Figure 3: Overview of our LEMIX framework.

task and `vTaskStartScheduler` to start the scheduler. Similarly, `xTimerCreate` function is used to register for a timer event.

Embedded systems use a portable layer enabling an RTOS to be used for different MCUs (Figure 1). As explained in § 2.2, most RTOSes maintain an NPL enabling them to run on top of regular OSes, *i.e.,* Windows (WPL) or Linux (LPL).

Given the source code of an embedded application, we identify the RTOS dependencies and re-configure them with corresponding LPL. To aid our process, we gather and maintain the LPL software packages of RTOSes apriori. This is not trivial as the build setup of the application may include MCU-specific configurations enabling certain HAL-specific APIs necessary for its functionality. For instance, in the FreeRTOS app TinyUSB, the `configTIMER_QUEUE_LENGTH` is set to 32, while the POSIX build sets it to 20, causing undefined behavior due to the application's expectation that this value should not exceed its configuration. In few cases, peripheral models implemented in the original RTOS may not be available in the corresponding LPL.

To address this, we designed a fully automated approach that selectively integrates configurations from the application that do not disrupt the LPL build. Each RTOS configuration from the application is iteratively toggled in the LPL build, retaining those that compile successfully. Once the LPL build successfully incorporates the necessary configurations, we replace the application's RTOS object files with those from the successful LPL build. We term this process as **POSIX Swap**. This approach however can induce unexpected behaviors in the ported application since not every configuration was incorporated from the application's config. But, we did not observe any false positive crashes due to misconfigured LPL build during our evaluation.

4.1.2 Interactive Resolution for Retargeting (Ch 2)

As mentioned in § 3.3, our goal is to build LEAPP for common desktop ISAs, specifically x86, because of the availability of various testing tools. We want to use the CLANG compiler as the LLVM IR enables us to easily perform various analysis tasks, and also, several techniques (*e.g.,* loop analysis) already exist in the CLANG infrastructure. However, just replacing the compiler with CLANG and changing the target ISA to x86 does not work. Because (as mentioned in § 3.3) embedded applications use non-standard C language features and inline assembly of other ISAs, *e.g.,* ARM. Handling this requires program semantic reasoning [45], a known hard problem.

We use an *interactive human-in-the-loop refactoring approach to tackle this*. We aim to automatically refactor the code to be CLANG and x86 friendly using a set of refactor patterns. However, for cases requiring semantic reasoning, we resort to developer assistance by providing precise guidance instructions. Our automation takes over after developer assistance, and the process continues with intermittent manual refactorings until the resulting code can be compiled using CLANG, *i.e.,* able to generate LLVM Bitcode. Table 12 (Extended Report [87]) summarizes automated and interactive refactorings. Further details of the build process tracing and streamlining the build system for x86-clang can be found in *Appendix B.2* of Extended Report [87].

We classify the set of refactorings into the following two categories and present the techniques used to handle them:

1. **Compiler Incompatibilities:** These are incompatibilities because of compilers (GCC v/s CLANG) and architecture-dependent code, *e.g.,* expecting **`int`** to be of 4 bytes. A significant portion of embedded software relies on GCC-based toolchains [92]. Hence, making the transition from a GCC build environment to CLANG is challenging, especially for embedded codebases [78]. Table 12 (Extended Report [87]) highlights the incompatibilities between GCC and CLANG affecting our embedded applications. Some of these, such as Variable-Sized Object initialization, are still not supported even in the latest version of LLVM at the time of writing (LLVM 18) [20]. Although several works [16, 78, 49] mention this problem, to the best of our knowledge, we are the first to highlight these issues, which have not yet received sufficient attention among embedded developers. Addressing compiler incompatibilities

requires semantically equivalent refactorings. We define a set of refactoring templates for automatically handling several of these issues and resort to developer assistance for others. We also provide guidance instructions to assist in the refactoring an example of which is shown in Listing 4 (Extended Report [87]).

2. **Inline Assembly:** Embedded applications often use inline assembly for low-level operations, such as MCU-specific initialization [78]. LEAPP eliminates the need for such initialization by relying on LPL. As discussed in § 3.2.2, precise handling of assembly is unnecessary for manifesting most vulnerabilities. We automate source code rewriting to identify and comment out inline assembly regions. Commenting out assembly may lead to uninitialized or undefined variables (*e.g.,* Listing 4 Appendix). Most inline assembly reads architecture-specific registers for initialization checks. To address this, we randomly initialize variables defined by assembly to 0 or 1, allowing applications to bypass initialization routines over multiple runs. Our approach may not handle all cases, such as inline assembly within macros or machine code representations (*e.g.,* Listing 5 in Appendix). In such cases, we automatically detect issues and provide developers with precise instructions, such as resolving compilation errors like *expected closing parenthesis*. LEMIX pinpoints problematic lines and suggests fixes, ensuring minimal manual intervention.

4.1.3 Opportunistic Symbol Resolution

The final step in creating the LEAPP involves linking compiled LPL and application object files. However, directly linking RTOS object files often results in linker errors [8] because embedded applications may invoke MCU-specific functions [78] that are not present in LPL, causing undefined reference errors [67]. For example, in the FreeRTOS application *Infinitime*, the function *xTimerGenericCommand* is invoked but not available in LPL, leading to a linker error.

LEMIX incrementally resolves linker errors by selectively linking only the required object files from the prebuilt LPL, resolving linker conflicts in an automated fashion. This is achieved using an opportunistic approach by identifying and linking MCU-specific RTOS object files (denoted as O_u) that define the missing symbols. However, this can cause multiple definition errors if symbols in O_u overlap with those in LPL. For instance, resolving *xTimerGenericCommand* by including the application kernel's *timers.o* introduces duplicate definitions, such as *prvInitialiseNewTimer*, between the application kernel's *timers.o* and the LPL kernel's *timers.o*. We want to ensure that references are linked with the expected symbols. We use a two-phase approach to tackle this:

1. **Creating Library Archives:** We observed that embedded applications and RTOSes use build procedures based on archiving [3], which prevents duplicate symbol errors across multiple components, *e.g.,* an application can have a function (say f) with the same function as an RTOS function. Creating an archive of application-specific object files ensures that all references to f within these object files are linked to the application-specific version. We trace the build process to identify which archives (and their order) were created and the corresponding object files. We follow the same order when creating and linking archive files to ensure that original references are intact.

2. **Modifying Symbol in One of the Objects:** The remaining multiple references cannot be resolved by archiving; hence, the symbol name has to be modified in one of the object files. If the multiple reference is between an Application's object file and LPL object file, we change the symbol name in the Application object file, which ensures that calls in the application to LPL functions are linked appropriately.

Handling Symbol Aliasing: In the original application's build procedure, the linker might create aliases for various symbols to maintain interoperability across different boards as guided by linker scripts. For instance, the symbol *__init_clock* might be resolved to aliases like *__stm32_clock_init* or *__nrf52_clock_init*, depending on the target board. Additionally, multiple aliases can be created for the same symbol. Ignoring such aliases results in `NULL`-ptr deferences while executing the resulting LEAPP. To tackle this, we first extract the aliasing information using firmware debug symbols from the original embedded application (compiled for the target MCU) and identify symbol aliases with the help of GDB. Next, we instrument our LEAPP and link the symbol aliases to the appropriate references in LEAPP.

4.1.4 Peripheral Modeling and Instrumentation (Ch 3)

As mentioned in § 2.3, our threat model includes malicious peripherals, *i.e.,* we assume that all inputs from peripherals can be controlled by an attacker. As mentioned in § 2.1, applications interact with peripherals by accessing corresponding MMIO addresses and interrupts.

Handling MMIO Accesses: Input from peripherals is received through reading MMIO addresses. We aim to model loads (*i.e.,* reads) from these addresses as reads from standard input and ignore stores (*i.e.,* writes) as we focus on vulnerability detection (*i.e.,* BMF as described in § 3.2.2).

First, we determine MMIO address ranges. One of the common techniques is to find these address ranges from peripheral System View Description (SVD) files [86]. However, as we will show in § 5.3, oftentimes, SVD files are incomplete and do not contain all peripheral address ranges which is also a known problem [4]. We aim to create an automated technique that does not depend on SVD files. Peripherals have predefined MMIO address ranges, and applications access them through hardcoded addresses [81, 23]. We perform a constant address analysis to determine all hardcoded address values,

```
uint32_t HAL_RCC_GetSysClockFreq(void) {
  uint32_t pllm = 0U, pllvco = 0U, pllp = 0U;
  uint32_t sysclockfreq = 0U;
  if (isMMIO(RCC->PLLCFGR)) {
  pllm = get_input_from_stdin() & RCC_PLLCFGR_PLLM;
  } else {
  pllm = RCC->PLLCFGR & RCC_PLLCFGR_PLLM; ⊟
  }
  ...
  sysclockfreq = pllvco / pllp; 🐞 ...}
```

Listing 2: ⊟ shows original MMIO accesses that are instrumented in LEAPP. 🐞 shows a div-by-zero bug in STM32.

i.e., constant values used as pointer operands in load and store instructions. The corresponding pages form the base MMIO pages (P_m). For instance, if we found a constant address x, then we will add the corresponding page $[b, b+4093]$ to P_m, where $b = (x\& \sim (0x3FF))$ is the base address of the corresponding page. We also perform additional coalescing and consider all pages within a range of ±2 KB from that boundary, also as MMIO addresses. This approach helps group related MMIO access and ensures that accesses within the same memory-mapped region are consistently recognized.

Next, we will hook all loads and stores through compile-time instrumentation and link with our runtime library. At runtime, our hook will check if a load is within an MMIO address range; if yes, then it will read an appropriate number of bytes from input and return the corresponding value. Similarly, our hook will ignore all stores to MMIO address ranges. Listing 2 shows the example of our instrumentation (highlighted lines), where the memory access `RCC->PLLCFGR` is checked to see if it is an MMIO address; if yes, we will read a value of corresponding size (*i.e.,* 4 bytes) from input.

Handling Interrupts: Interrupts are treated as peripheral inputs and triggered at random intervals. LEMIX identifies Interrupt Service Routines (ISRs) using RTOS-specific patterns, such as ISR vector tables in assembly files, while ignoring handlers implemented in assembly. Using RTOS-specific templates, we create a *Dispatcher Task* to invoke ISRs at arbitrary intervals (Listing 7 in Extended Report [87]). To prevent false crashes from ISRs requiring preconditions (e.g., valid global pointers), we use lightweight binary analysis and dynamic tracing to identify and disable them.

4.1.5 Dynamic Analysis Assistance Instrumentation

Embedded applications have considerable peripheral state-dependent code [71]. Specifically, they check for peripherals to be in a specific state before interacting with it or to perform some interesting function, *e.g.,* as shown in Listing 3, at lines 3-4, the code busy-waits until the control register (accessed through MMIO read `NRF_CLOCK->LFCLKSTAT`) has any of the bits corresponding to `CLOCK_LFCLKSTAT_STATE_Pos` that are not set.

Peripherals state is accessed through reading certain registers [42, 71], *e.g.,* clock state is accessed through `NRF_CLOCK->LFCLKSTAT` (an MMIO address) in Listing 3. Since

```
1   // Before Condition Weakening
2   while
3       ((NRF_CLOCK->LFCLKSTAT & CLOCK_LFCLKSTAT_STATE_Pos)⊘) {
4       // Busy Waiting
5       ⁘ };
6   interesting_function();
7   // After Condition Weakening
8   label:
9       bool cond = (NRF_CLOCK->LFCLKSTAT &
10                      CLOCK_LFCLKSTAT_STATE_Pos); ⊘
11      new_cond = cond;
12      if (isMMIO(NRF_CLOCK->LFCLKSTAT) && stdin_read() % 2) 🔧
13          new_cond = !cond; ⚭
14      while (new_cond) {goto label};
15  interesting_function();
```

Listing 3: ⊘ indicates MMIO coverage blockers, ⁘ marks busy waiting due to unsolved constraints, 🔧 represents our instrumentation and ⚭ shows MMIO condition toggling.

we model all peripheral reads (§ 4.1.4) as reads from standard input, the coverage of state-dependent code becomes the problem of constraint input generation. For instance, in Listing 3, the MMIO access, *i.e.,*, `NRF_CLOCK->LFCLKSTAT` will be fetched from input. For the execution to reach out of the loop, an input generation technique (*e.g.,* AFL++) should provide an input that satisfies the constraint. Existing techniques handle this by providing precise peripheral models [71, 81] or symbolic execution [19], but they have scalability issues [22].

To tackle this, we perform *Weakening of Peripheral State Dependent conditions*. Specifically, we instrument each conditional instruction to check whether it involves reading from an MMIO address; if yes, we weaken the condition such that any value can satisfy the constraint with 50% probability as shown in the lower part of Listing 3. Previous works [60, 14] show that such an approach improves the effectiveness of fuzzing. We will also show in § 5.5 that our approach greatly improves the coverage. We also perform instrumentation to collect additional metrics, such as coverage.

4.2 Phase 2: Testing

This phase focuses on fuzzing of LEAPPS generated in Phase 1. We explore two modes of fuzzing: (i) Whole program and (ii) Function level.

4.2.1 Whole Program Fuzzing

Here, we fuzz LEAPPS as a whole by providing inputs at appropriate locations (*i.e.,* MMIO accesses) until LEAPP terminates or crashes because of a bug.

4.2.2 Function Level Fuzzing

In this mode, we directly fuzz individual functions by providing arguments of appropriate type [101, 6]. Given a function f, we use a simple co-relation analysis [62] to determine the argument types and their size associations, *e.g.,* for a function that accepts two arguments, an integer array

Table 2: Approximate Number of unique basic blocks discovered by various configurations of LEMIX in comparison to State Of The Art Tools (Discussed in § 5.4 and § 5.6). M1- M3 represents different configuration modes for LEMIX. Refer to Table 13 (Extended Report [87]) for a larger version.

AppID	Lx						Fw			Mf		
	M1		M2		M3							
	Cov	Bug	Cov	Bug	Cov	Bug	Cov	Crash	Bug	Cov	Crash	Bug
f1	731	0	2.9k	1	</>	1	500	1	0	1k	0	0
f2	456	1	2.8k	3	</>	1	❗	N/A	N/A	1.5k	1	1
f3	560	1	668	2	1.5k	3	❗	N/A	N/A	❗	N/A	N/A
f4	563	0	1k	1	6k	3	500	0	0	2k	41	0
f5	442	0	728	2	1.8k	2	700	0	0	1.8k	93	0
n1	105	0	301	1	13.5k	1	356	0	0	25.2k	148	0
n2	143	0	338	0	16.8k	1	405	0	0	300	0	0
n3	157	1	357	1	19.9k	1	60	1	1	100	1	0
n4	135	0	235	1	19.5k	1	388	3	0	2.4k	0	0
n5	823	0	1.5k	1	</>	0	134	0	0	226	0	0
z1	76	0	86	2	3k	4	44	2	0	213	0	0
z2	🟦	0	🟦	2	12.3k	3	❗	N/A	N/A	10.8k	483	1
z3	🟦	0	🟦	2	4.6k	6	❗	N/A	N/A	❗	N/A	N/A
t1	210	0	553	0	6.3k	0	670	0	0	750	0	0
t2	214	0	553	0	240	0	791	0	0	694	0	0
t3	🟦	0	🟦	0	3.1k	0	900	0	0	700	0	0
t4	310	0	455	1	188	1	749	0	0	659	10	1
t5	254	0	436	0	5.2k	0	748	0	0	658	0	0
Avg/Tot	345	3	860	20	7.5k	28	496	7	1	3.1k	777	3
Unique Bugs		3		10		11			1			3

pointer, and its length; our co-relation analysis will produce: `{arg1: {ARRAY, int, SIZE: arg2}, arg2: int}`. Next, we automatically create generators for each of the argument types, *i.e.,* functions that generate values of a specific type from the input. For instance, for the above example, the generator will create an integer array of an arbitrary size, populate it with random integer values, and return the pointer and size. Finally, we invoke f with the pointer returned by the generator and the size as the second argument. Unlike whole program fuzzing (§ 4.2.1), each fuzzing run in function level fuzzing invokes the target function once and exits.

5 Evaluation

We use a combination of Python scripts and CLANG/LLVM 10 toolchain passes to implement our framework. We provide more details in *Appendix B.2* of Extended Report [87]. We evaluate LEMIX by answering the following questions:

RQ1 (*Converting to* LEAPP *(§ 5.2)*): How effective is our approach (§ 4.1.2) in converting embedded applications to LEAPPs? How much manual effort does it require?
RQ2 (*Peripheral Handling (§ 5.3)*): How effective is our approach (§ 4.1.4) in identifying MMIO addresses?
RQ3 (*Testing* LEAPP *Applications (§ 5.4)*): What is the effectiveness of testing LEAPPs through different fuzzing approaches, *i.e.,* whole-program fuzzing and function-level fuzzing?
RQ4 (*Ablation Study (§ 5.5)*): What is the contribution of our peripheral handling (§ 4.1.4) and dynamic analysis assistance (§ 4.1.5) on overall effectiveness?
RQ5 (*Comparative Evaluation (§ 5.6)*): What is the effectiveness of LEMIX compared to the existing state-of-the-art?
RQ6 (*False Positive Analysis (§ 5.7)*): What false positives are introduced by low-fidelity execution, how do they compare against existing State-Of-The-Art, and how are they remediated?

RTOS	ID	SRC	ASM	RTOS	SDK
FreeRTOS	f1	88k	2.2k	105k	100k
	f2	22k	2.6k	13.5k	2.16M
	f3	32.4k	1k	10.3k	130k
	f4	657k	1.5k	209k	65k
	f5	656k	2k	209k	
Nuttx	n1	429k	26k	1.7M	8k
	n2	428k	22k	1.6M	
	n3	429k	23k	1.65M	
	n4	429k	25k	1.7M	
	n5	310k	600	1.5M	198k
Zephyr	z1	200	1k	19k	0
	z2	5.6k	0	20k	3k
	z3	14.2k	0	20k	2.4k
Threadx	t1	413k	52.1k	351k	335k
	t2	236k	52.2k	351k	
	t3	333k	51.3k	351k	
	t4	185k	51.4k	351k	
	t5	310k	52k	351k	

Table 3: Breakdown of Source Lines of Code by source files, assembly (inline + standalone) , RTOS, and SDK (counted once per application). See Table 7 in Extended Report [87] for descriptions of each application.

5.1 Dataset and Setup

Dataset: Table 3 gives details of our application set with per-component SLOC, selected in 2 steps.

First, to study Type-2 embedded systems, we ensured diversity at the RTOS level, choosing four popular RTOSes: FreeRTOS (widely used in resource-constrained systems), Zephyr (modularity and scalability), Nuttx (POSIX-compliant and versatile), and ThreadX (optimized for high-performance real-time applications).

Second, we sampled applications for each RTOS from GitHub, including major, actively maintained projects (e.g., PX4 for drones, Infinitime for smartwatches) and smaller, peripheral-focused ones (e.g., TinyUSB for USB, Nrf_Pwm for PWM tasks).

Setup: We have conducted our experiments on an AMD EPYC 7543P 32 Core Processor with 64 threads and 128 GB of RAM. In whole program mode, we fuzzed each application for 48 hours following suggested best practices [37]. In function-level mode, we fuzzed each target function for 10 minutes, after which coverage plateaued for most functions.

5.2 RQ1: Converting to Linux Applications

5.2.1 Methodology

We measure the ability of LEMIX to successfully convert the 18 embedded applications in our dataset to Linux applications and the amount of manual effort required. An application is successfully converted if it can be compiled and executed on a Linux operating system without crashing. Additionally, as described in § 4.1.2, LEMIX relies on human intervention to guide retargeting to desktop ISA. We categorize the required human effort into three categories as follows: (a) Setup (Identifying source files and build/compilation instructions), (b) Addressing errors due to compiler incompatibilities, and (c) Handling inline assembly. We measure the time spent in each category and the impact on the application's source files.

These conversions were performed by the authors, who are graduate students with intermediate expertise in C/C++ but with less experience with embedded codebases. The conversion time to LEAPP depends on familiarity with the embedded codebase, so the reported measurements represent an upper bound; engineers with embedded expertise should require significantly less time.

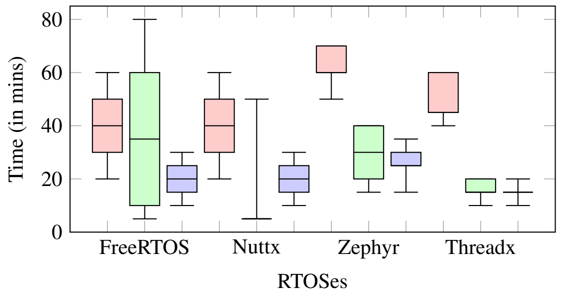


Figure 4: Comparison of manual effort time (y-axis) across RTOSes (x-axis), categorized into three categories: (a) 🟥 - Preconfigurations and source modifications; (b) 🟩 - Non-automated compiler errors; (c) 🟦 - Macro ASM adjustments.

5.2.2 Results

Using LEMIX's interactive approach, we successfully converted all the applications in our dataset to LEAPPs. As shown by the results in Table 2, we were able to execute and dynamically analyze all 18 applications.

Figure 4 shows the box plot of time spent in each human effort category for applications across different RTOSes. In all RTOSes, setup time (category (a), median 40–60 min) is the largest contributor, mainly for identifying the build setup, dependencies, and toolchains. These times align with Shen *et al.,* [77], who reported an average of 60 min for embedded build setup. Note that setup time is independent of LEMIX. As shown in Figure 4, our interactive steps (categories (b) and (c)) contribute minimally to the manual effort.

Table 4 highlights SLoC affected for categories (b) and (c). Manual effort for (b) is relatively low (median 20–40 min) compared to the SLoC modified, demonstrating effective handling of compiler incompatibilities. For example, NuttX applications required no manual effort for (b) as they used standard C features supported by CLANG, larger FreeRTOS applications required more effort due to greater SLoC changes.

Despite of a large number of ASM modifications, the amount of manual effort (*i.e.,* category (c) with a median of 20 - 30 min in Figure 4) is relatively less, demonstrating the effectiveness of our source code transformations to automatically handle inline assembly.

RQ1 results demonstrate that LEMIX can successfully convert embedded applications to LEAPPs and requires minimal manual effort.

5.3 RQ2: Peripheral Handling

5.3.1 Methodology

We assess the effectiveness of our constant address analysis in two ways. First, we validate discovered address ranges by checking for overlaps with the LEAPP's actual memory map, ensuring MMIO ranges remain distinct. This validation leverages standard memory boundaries documented in SVD files, which define peripheral registers and their address mappings.

Second, we corroborate the results of LEMIX's constant address analysis by comparing the discovered address ranges against those specified in CMSIS-SVD files [54]. Discrepancies are manually investigated through random sampling to understand gaps in identification. Both methods are necessary to ensure accuracy and to identify limitations of SVD-based documentation versus our constant address analysis approach.

5.3.2 Results

The Table 5 shows the number of MMIO address ranges found across different applications. Upon investigation, we found that none of these address ranges conflict with the memory map of the corresponding LEAPP. Hence, instrumenting reads from these addresses should not affect LEAPP's execution.

When we compared the discovered address ranges with those in CMSIS-SVD files [54], we found that over 50% of our address ranges are missing in CMSIS-SVD files (last column of Table 5). Further analysis revealed that the missing address ranges represented valid MMIO addresses in the source code and corresponded with those used by valid core peripherals [4]. Listing **??** (Appendix) shows MMIO address ranges used in the codebase but missing from the corresponding peripheral's SVD file.

Table 4: Detailed porting metrics for each application, including type of files modified, lines added or removed, and impact percentages (lines affected over total lines). The times are summarized in Figure 4.

AppID	Total Files		Total Lines		Files Modified			Lines Added/Removed		Impact %
	App	RTOS	App	RTOS	Sources	Headers	ASM	Sources + Headers	ASM	
								Category (b)	Category (c)	
f1	1.1k	612	184k	105k	2	8	6	2, -2	+505, -964	0.51
f2	5.8k	100	2.38M	13.5k	2	3	11	+218, -6	+410, -1483	0.09
f3	201	31	162.5k	10.3k	10	5	4	+502, -128	+539, -953	1.23
f4	232	584	724k	209k	3	2	4	+239, -2	+539, -1007	0.19
f5	230	584	723k	209k	2	1	4	+238, -2	+539, -1007	0.19
n1	400	14k	429k	1.7M	0	0	4	NIL	+133, -500	0.03
n2	378	13.9k	428k	1.6M	0	0	4	NIL	+133, -500	0.03
n3	355	13.8k	429k	1.65M	0	0	7	NIL	+147, -537	0.03
n4	400	14k	429k	1.7M	0	0	8	NIL	+148, -575	0.03
n5	455	12.9k	435k	1.5M	3	2	1	+6, -6	+152, -400	0.03
z1	50	6.2k	200	20k	7	3	0	+235, -171	NIL	1.27
z2	203	6.2k	8.6k	20k	7	3	0	+220, -180	NIL	1.40
z3	221	6.2k	16.6k	20k	7	3	0	+220, -180	NIL	1.09
t1	4.8k	1.3k	748k	351k	1	0	2	+3, 0	+568, -884	0.14
t2	3.8k	1.3k	571k	351k	0	0	2	NIL	+569, -879	0.16
t3	3.5k	1.3k	668k	351k	1	0	2	+4, 0	+568, -884	0.14
t4	3.3k	1.2k	520k	351k	0	0	1	NIL	+23, -167	0.02
t5	4.3k	1.2k	645k	351k	1	0	2	+3, 0	+568, -884	0.14

RQ2 results show that our constant address analysis is effective at finding MMIO address ranges and provides more complete results than the commonly used approach of analyzing SVD files.

5.4 RQ3: Testing LeApps

5.4.1 Methodology

In this RQ, we assess the effectiveness of the converted LeApps in supporting different fuzzing modes: Whole Program Fuzzing with MMIO instrumentation (M1), Whole Program Fuzzing with MMIO + Weakening State-Dependent Conditions (M2), and Function-Level fuzzing (M3) incorporating all optimizations from (M1) and (M2). We measure and report the code coverage (in terms of unique basic blocks covered) and the number of unique bugs detected through each mode, following crash triaging and manual confirmation according to our threat model. For whole-program fuzzing, we identified the MCU firmware ELF entrypoint and ensured the LeApp's entrypoint matched it (ignoring assembly-based entrypoints). This was necessary to initialize global structures for peripheral handling and avoid `NULL`-ptr dereferences in the LeApp. To identify candidate functions for function-level fuzzing, we first filtered functions that take pointer arguments without a specified size. Next, we manually verified (5 minutes per function) whether these functions performed any interesting operations, such as pointer arithmetic or explicit casts, which are common in risky programming idioms. Previous work [21] indicates that these characteristics are strong indicators of potentially buggy functions. Depending on the target, this process typically leaves us with roughly 100-150 functions per application for further fuzzing.

5.4.2 Results

Table 2 shows the code coverage and bug detection results when conducting whole program ($M2$) and function-level ($M3$) fuzzing. Using Lemix, we conducted whole program and function-level fuzzing on 15 applications each. We manually created the memory layout for z1 as a demonstration but did not perform whole-program fuzzing for z2, z3, and t3 due to their layout-dependent code, which is not automated (§ 6). We did not perform function-level fuzzing on f1, f2, and n5 as they were written in C++, and our current implementation of function-level fuzzing does not support C++ objects.

Code Coverage: In whole program fuzzing, we triggered a considerable number of reachable functions (*i.e.,* those that can be reached through `main`) in each LeApp. Figure 5 shows the percentage of triggered functions, with over 70% triggered on average, except for f2, f3, and z1. The Cumulative Distribution Function (CDF) in Figure 5 illustrates function coverage, where each point (x,y) indicates that x% of triggered functions have y% or less code coverage. The consistent slope across LeApps confirms that Lemix enables effective testing with reasonable coverage. For example, in FreeRTOS LeApps, 40% of triggered functions achieve 40% or more code coverage. Table 2 shows absolute coverage, with function-level fuzzing providing ∼10x more coverage than whole-program fuzzing, as it targets individual functions.

AppID	Detected MMIOs	In SVD (% of Detected)
f1	45	11 (24.44)
f2	33	16 (48.48)
f3	35	18 (51.43)
f4	15	11 (73.33)
f5	15	11 (73.33)
n1	8	4 (50.0)
n2	10	5 (50.0)
n3	9	4 (44.44)
n4	9	4 (44.44)
n5	60	9 (15.0)
z1	10	4 (40.0)
z2	10	5 (50.0)
z3	54	25 (46.3)
t1	16	6 (37.5)
t2	16	6 (37.5)
t3	3	1 (33.33)
t4	16	6 (37.5)
t5	16	6 (37.5)

Table 5: MMIO detection analysis highlights potential undocumented peripherals in SVD files. SVD Detection shows documented MMIOs, while Potential MMIOs indicates detected MMIOs that may represent undocumented peripherals.

Bug Detection: Table 2 also shows the bugs detected by each approach. Overall, as expected, function level fuzzing ($M3$) identified 11 additional unique bugs. This is due to its ability to directly exercise risky functions. As shown in Listing 1, function-level fuzzing ($M3$) uncovered an out-of-bounds access in a deep function that whole-program fuzzing ($M2$) missed, as the function was never triggered.

As shown in the last row of Table 2, although total bugs are large (*e.g.,* 28), the number of unique bugs is small (*e.g.,* 11). This is because the same bugs (those in RTOS functions) could be present multiple in LEAPPS. More details can be found in *Appendix F.1* of Extended Report [87]. Table 7 (Appendix) summarizes the bug types, affected applications, bug descriptions, and developer responses. The Table 9 (Extended Report [87]) shows a detailed split of bugs and unique bugs. The Table 10 (Extended Report [87]) provides the categorization of unique bugs. We found several memory corruption bugs in addition to robustness bugs, such as Divide by zero (Listing 2).

RQ3 results demonstrate that LEMIX facilitates whole-program and function-level fuzzing, leading to high code coverage and bug detection.

5.5 RQ4: Ablation Study

5.5.1 Methodology

This RQ measures the contributions of our peripheral handling (§ 4.1.4) and condition weakening (§ 4.1.5) instrumentation in facilitating dynamic analysis. We disabled each of these instrumentations and report their impact on whole-program fuzzing. While function-level fuzzing performed better, whole-program fuzzing compensated for the limitations of function-level fuzzing on C++ applications and provided insights into how effectively a LEAPP can be tested as a standalone application.

5.5.2 Results

Peripheral handling instrumentations: When the instrumentations on MMIO accesses are disabled, we observed that all LEAPPS crash immediately after they are started. As mentioned before, LEAPPS are fuzzed as regular Linux applications, in which MMIO addresses may not be mapped; consequently, any MMIO accesses will result in invalid memory access and segfault. This shows that *our peripheral handling instrumentation is necessary for testing* LEAPP*s.*

Condition weakening instrumentation: When the instrumentations that weaken state-dependent conditions are disabled, we observe a remarkable drop in the number of covered basic blocks. The **M1** and **M2** columns in Table 2 shows the number of covered basic blocks and bugs found when conducting whole-program fuzzing without and with this instrumentation. On average, we see an improvement of ∼2x in the number of basic blocks covered with M2 over M1. All bugs found by M1 were also detected by M2, with the addition of 7 more bugs. These results show that embedded applications greatly depend on the peripheral state for their execution, and ignoring them results in ineffective testing.

RQ4 results show that our instrumentation-based techniques significantly improve the effectiveness of testing.

5.6 RQ5: Comparative Evaluation

5.6.1 Methodology

In this RQ, we compare the code coverage and bug detection results of LEMIX with results from other recent dynamic analysis techniques that target embedded applications. We selected baselines that follow the LEMIX's philosophy of being usable on applications without requiring low-level understanding of the application's internal implementation. This led to three baselines: P2IM [23], Fuzzware (Fw) [71] and MultiFuzz (Mf) [13], and the exclusion of three others: PMCU [44], Halucinator [15] and SAFIREFUZZ [72]. Notably, PMCU required a custom RTOS configuration for each application based on the application's internal implementation. Halucinator and SAFIREFUZZ require creating handlers for each peripheral the application interacted with.

We were unable to set up P2IM, despite following their instructions and attempting to contact the authors. Additionally, we encountered challenges setting up fuzzing for certain applications (*e.g.,* f3, z3) using Fuzzware and Multifuzz,

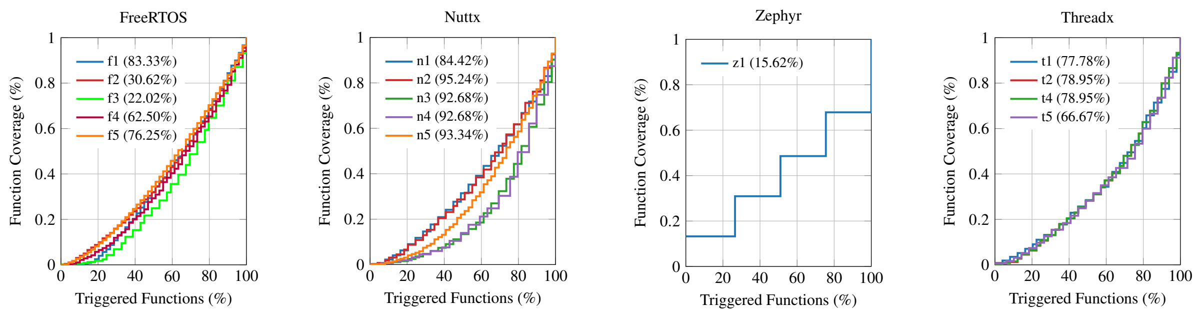


Figure 5: Cumulative Distribution Functions (CDFs) of function coverage across various applications. Each line represents an application's coverage distribution in app-level fuzzing, illustrating the proportion of functions (x-axis) that achieve at most a given coverage percentage (y-axis).

primarily due to inaccurate memory modeling, resulting in applications crashing with unsupported or invalid instructions. Consequently, we evaluated Fuzzware and Multifuzz on the remaining original, non-converted LEAPPS.

5.6.2 Results

Table 2 shows the results of fuzzing each LEAPP with the selected baselines. We found that, on average, LEMIX configurations (M2/M3) detected 21 bugs, while Multifuzz (*Mf*) and Fuzzware (*Fw*) detected 1 and 3 bugs, respectively. From a code coverage perspective, *Mf* outperformed *Fw* for most applications. However, LEMIX configurations (M2/M3) outperformed *Mf* for most applications except for n1 and z2. This was primarily due to *Mf*'s ability to trigger nested interrupts, which led to higher coverage. In contrast, LEMIX uses a simpler interrupt handling approach (§ 4.1.4) and does not support nested interrupts. We also observed that *Fw* occasionally reported false positives, such as crashes in z1, caused by incorrect interrupt handling. From the bug detection perspective, LEMIX is even more effective by detecting 21 bugs, with Fuzzware and MultiFuzz detecting only 1 and 3 respectively. Furthermore, all bugs found by Fuzzware and MultiFuzz are also found by LEMIX.

RQ5 results indicate that LEMIX has better bug-finding ability than existing techniques.

5.7 RQ6: False Positive Analysis

5.7.1 Methodology

This RQ aims to analyze false positives that arise due to our low-fidelity approximations, how they compare with existing works, and provide remediation for each type of false positive. Our analysis covers the LEMIX components that introduce approximations, as these are the sources of false positives.

AppID	Inline ASM	Interrupt Misfiring	Board Layout
f1	1	1	0
f2	1	3	0
z2	0	0	1
z3	0	0	1
t2	0	1	1
Total	**2**	**5**	**3**

Table 6: False positives due to LEMIX approximations.

5.7.2 Results

Table 6 summarizes the number and type of false positives encountered across various applications.

POSIX Swap (§ 4.1.1): LEMIX replaces the board-specific RTOS layer with a POSIX-compatible one. While this can introduce false positives due to kernel misconfiguration, we observed none in our case. For example, incorrect task priorities could affect behavior, but we mitigated this by incorporating all relevant application kernel configurations.

Inline ASM (§ 4.1.2): LEMIX removes all inline assembly and approximates expected values at runtime using random values of the same type This led to two false positives across all applications. For instance, Listing 4 (Appendix) shows how inline assembly was used for initialization, which we identified through debugger-inspected halts. We resolved this by manually patching the instruction to return the expected value.

Symbol Modifications (§ 4.1.3): Our symbol modification strategy iteratively resolves linker errors and could, in principle, introduce false positives such as from incorrect renaming of indirect function calls. However, we observed none.

Interrupt Misfiring *(§ 4.1.4)*: LEMIX attempts to trigger all interrupts from the *isr_vector_table*, which can cause crashes if global structures containing callback routines are uninitialized. For example, Listing 9 of Extended Report [87] shows a misfired interrupt caused by this dependency. We remediate using lightweight static analysis to trigger interrupts accurately.

Board Layout (§ 6): Board-specific layout-dependent code is a limitation of our work, preventing analysis of two Zephyr and one ThreadX applications. An example of this issue is shown in Listing 6. We manually constructed the layout for one Zephyr application for our evaluation.

The existing tools used in our Comparative Evaluation (§ 5.6) also suffer from false positives due to misfired interrupts and emulation issues. In contrast to LEMIX, these tools had a greater number of false positive crashes, as indicated by the large numbers along the crash column of Table 2. Furthermore, triaging these crashes (especially in the case of Fuzzware (*Fw*)) is non-trivial, and has also been acknowledged by recent work [10]. LEMIX adopts an approximate but principled approach, enabling us to easily identify false positives and resolve them.

RQ6 results indicate that the lower-fidelity execution model used by LEMIX does not lead to a large number of false positives, as evidenced by our comparative evaluation.

6 Limitations and Future Work

We recognize the following limitations of LEMIX and plan to handle them as part of our future work.

- **Dependency on LPL:** Our approach depends on the existence of LPL for RTOSes. As shown in *Appendix B.0.3 of Extended Report [87]*, most RTOSes already have LPL, we argue that it is fairly easy to create LPL based on existing implementations.
- **Incomplete ISR coverage:** Our approach identifies ISRs via RTOS-specific pattern matching, which worked reliably in our experiments. However, we skip ISRs that depend on global state, leading to some coverage gaps. Recent works like AIM [24] improves ISR identification and invocation, and we plan to incorporate such techniques into LEMIX in future work.
- **Unable to handle layout-specific code:** We found cases where embedded applications rely on specific memory layouts, hindering our efforts in further analysis. Listing 6 (Appendix) shows an example from a Zephyr RTOS application. As future work, we plan to automatically detect and refactor such code idioms.

7 Related Work

Dynamic analysis techniques, especially automated random testing or fuzzing [51, 29], are demonstrated to be effective at vulnerability detection. *Rehosting* is a necessary requirement for scalable dynamic analysis. This process is relatively easy for Type-1 systems [11, 46], *i.e.,* those based on standard OSes such as Embedded Linux. Consequently, several techniques [22] exist for rehosting Type-1 systems. But, these cannot be applied to Type-2 systems because of lack of well-defined OS interface and tight coupling with hardware [22].

One of the most important challenges of Rehosting Type-2 systems is the capability to handle peripheral interactions. Existing techniques to handle this can be categorized at a high level into hardware-in-the-loop [34] or software model [81] based approaches. The hardware-in-the-loop approaches [55, 43, 68, 30, 17, 33, 34, 38] achieve the highest level of fidelity and less manual effort. Given the diversity of hardware platforms, these techniques are hard to scale.

The software-only approaches [23, 103, 71, 35, 83] provide low-fidelity execution unless there are precise peripheral models. Automated peripheral modeling techniques [15, 23, 81, 31] are specific to certain peripherals and hard to generalize. Some techniques [71, 17] use symbolic execution [36] to create peripheral models. As shown by the recent systematization work [22], these techniques are hard to extend for different peripherals and depend on the existence of emulators [63, 50] of the corresponding ISA. On the other hand, works such as METAEMU [12] attempt to rehost firmware in an architecture-agnostic way by lifting firmware code to an Intermediate Representation as directed by Ghidra's Language Specifications [66] enabling multi-target analysis. However, these techniques struggle with manual efforts required for specification creation, peripheral modeling, and limited support for specialized automotive protocols.

One of the most closely related works is by Li *et al.,*[44], who rehost MCU libraries for testing on Linux by implementing a portable MCU (PMCU) using the POSIX interface and abstracting hardware functions. However, their method relies on hand-written abstractions for specific libraries and does not handle unknown or undocumented peripherals, nor does it scale well across diverse firmware binaries. LEMIX side-steps the problem of precise peripheral emulation by using NPL, which relaxes the requirement of precise peripheral models without affecting the execution of the target embedded system. Unlike prior works that require accurate peripheral models or emulation for specific hardware targets, our approach generalizes across firmware by focusing on what is sufficient to trigger bugs, rather than replicating exact hardware behavior. As a result, dynamic analysis can be applied to embedded code in a more generalizable manner.

8 Conclusion

We propose LEMIX, a novel approach to rehosting embedded applications as Linux applications by providing solutions to the associated challenges of retargeting to X86, preserving the execution semantics, and handling the peripheral interactions. We evaluated LEMIX on 18 embedded applications across four RTOSes and found 21 previously unknown bugs, most of which are confirmed and fixed by the corresponding developers. Our comparative evaluation shows that LEMIX outperforms existing state-of-the-art techniques in testing embedded applications.

9 Ethics Considerations

This section outlines the ethical considerations in designing, implementing, and evaluating LEMIX. The primary stakeholders are:

- Maintainers of the evaluated RTOSes and embedded applications.
- Users of those systems.

Risks and Benefits from Discovered Bugs: Our evaluation of LEMIX resulted in 21 new bugs. These defects, though not deemed security-critical, could lead to malfunctions or crashes. Hence, we reported all issues via public bug trackers and submitted corresponding patches. Most were acknowledged and merged by maintainers. Users who do not update may face residual risks if bugs are later exploited.

Risks and Benefits from LEMIX's Release: LEMIX will be open-sourced (see §10), making it available to both defenders and potential attackers. Like other defect discovery tools (e.g., fuzzers), it can be used ethically or maliciously.

These risks are common in vulnerability research. Our work aligns with accepted ethical standards in cybersecurity. To minimize risk, we submitted patches with each report; vendors did not treat the bugs as significant security threats, and no CVEs were issued.

10 Open Science

We have released a raw development version of LEMIX along with the dataset and necessary documentation at : https://zenodo.org/records/15611391 and https://seemoss.org/.

11 Acknowledgments

We would like to thank the reviewers and our shepherd for their valuable comments and inputs to improve our paper. This research was supported by Rolls-Royce and the National Science Foundation (NSF) under Grant CNS-2340548. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of Rolls-Royce and NSF.

References

[1] Omar Alrawi et al. "SoK: Security Evaluation of Home-Based IoT Deployments". In: *2019 IEEE Symposium on Security and Privacy* (2019).

[2] Manos Antonakakis et al. "Understanding the mirai botnet". In: *26th USENIX security symposium (USENIX Security 17)*. 2017.

[3] *ar(1) - Linux man page*. URL: https://linux.die.net/man/1/ar (visited on 01/16/2025).

[4] ARM-software. *SVD files: Missing the Core Peripherals · Issue #844 · ARM-software/CMSIS_5*. https://github.com/ARM-software/CMSIS_5/issues/844. Accessed: 2025-01-20.

[5] Miguel A Arroyo. "Bespoke Security for Resource Constrained Cyber-Physical Systems". en. In: *ProQuest Dissertations and Theses*. Accessed 15 Feb. 2023. Columbia University, 2021, p. 171. URL: https://www.proquest.com/dissertations-theses/bespoke-security-resource-constrained-cyber/docview/2470276679/se-2?accountid=13360..

[6] Domagoj Babić et al. "FUDGE: Fuzz Driver Generation at Scale". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019.

[7] Roberto Bagnara, Abramo Bagnara, and Patricia M Hill. "The MISRA C coding standard and its role in the development and analysis of safety-and security-critical embedded software". In: *International Static Analysis Symposium*. 2018.

[8] Katharina Bogad and Manuel Huber. "Harzer Roller: Linker-Based Instrumentation for Enhanced Embedded Security Testing". In: *Proceedings of the 3rd Reversing and Offensive-Oriented Trends Symposium*. 2020.

[9] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. "Fuzzing: Challenges and Reflections." In: *IEEE Software* 38.3 (2021), pp. 79–86.

[10] Boyu Chang et al. " FirmRCA: Towards Post-Fuzzing Analysis on ARM Embedded Firmware with Efficient Event-based Fault Localization ". In: *2025 IEEE Symposium on Security and Privacy (SP)*. 2025.

[11] Daming D. Chen et al. "Towards Automated Dynamic Analysis for Linux-based Embedded Firmware". In: *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016.

[12] Zitai Chen, Sam L Thomas, and Flavio D Garcia. "Metaemu: An architecture agnostic rehosting framework for automotive firmware". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2022.

[13] Michael Chesser, Surya Nepal, and Damith C. Ranasinghe. "MultiFuzz: A Multi-Stream Fuzzer For Testing Monolithic Firmware". In: *33rd USENIX Security Symposium (USENIX Security 24)*. 2024.

[14] Jaeseung Choi et al. "Grey-box concolic testing on binary code". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019.

[15] Abraham A. Clements et al. "HALucinator: firmware rehosting through abstraction layer emulation". In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020.

[16] Jake Corina et al. “Difuze: Interface aware fuzzing for kernel drivers”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017.

[17] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. “Inception: System-Wide Security Testing of Real-World Embedded Systems Software”. In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[18] Daniele Cono D’Elia et al. “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*. 2019.

[19] Drew Davidson et al. “FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution”. In: *22nd USENIX Security Symposium (USENIX Security 13)*. 2013.

[20] dgookin. *Not Every Compiler Likes Your Code | C For Dummies Blog*. Jan. 2023. URL: https://c-for-dummies.com/blog/?p=5711.

[21] Xiaoning Du et al. “Leopard: Identifying vulnerable code for vulnerability assessment through program metrics”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019.

[22] Andrew Fasano et al. “Sok: Enabling security analyses of embedded systems via rehosting”. In: *Proceedings of the 2021 ACM Asia conference on computer and communications security (AsiaCCS)*. 2021.

[23] Bo Feng, Alejandro Mera, and Long Lu. “P2IM: Scalable and Hardware-independent Firmware Testing via Automatic Peripheral Interface Modeling”. In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020.

[24] Bo Feng et al. “AIM: Automatic Interrupt Modeling for Dynamic Firmware Analysis”. In: *IEEE Transactions on Dependable and Secure Computing* (2024).

[25] *FreeRTOS*. http://freertos.org.

[26] Jian Gao et al. “EM-Fuzz: Augmented Firmware Fuzzing via Memory Checking”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2020).

[27] Mohammed Ali Al-Garadi et al. “A Survey of Machine and Deep Learning Methods for Internet of Things (IoT) Security”. In: *IEEE Communications Surveys & Tutorials* (2020).

[28] Vahid Garousi et al. “Testing embedded software: A survey of the literature”. In: *Information and Software Technology* (2018).

[29] Patrice Godefroid. “Fuzzing: Hack, art, and science”. In: *Communications of the ACM* (2020).

[30] Zhijie Gui et al. “Firmcorn: Vulnerability-oriented fuzzing of iot firmware via optimized virtual execution”. In: *IEEE Access* 8 (2020). Publisher: IEEE, pp. 29826–29841.

[31] Eric Gustafson et al. “Toward the Analysis of Embedded Firmware through Automated Re-hosting”. In: *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. 2019.

[32] Thomas A Henzinger and Joseph Sifakis. “The embedded systems design challenge”. In: *FM 2006: Formal Methods: 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006. Proceedings 14*. 2006.

[33] Markus Kammerstetter, Daniel Burian, and Wolfgang Kastner. “Embedded security testing with peripheral device caching and runtime program state approximation”. In: *10th International Conference on Emerging Security Information, Systems and Technologies (SECUWARE)*. 2016.

[34] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. “Prospect: peripheral proxying supported embedded code testing”. In: *Proceedings of the 9th ACM symposium on Information, computer and communications security*. 2014.

[35] Mingeun Kim et al. “FirmAE: Towards Large-Scale Emulation of IoT Firmware for Dynamic Analysis”. In: *Annual Computer Security Applications Conference* (2020).

[36] *KLEE*. URL: http://klee.github.io/ (visited on 02/07/2023).

[37] George Klees et al. “Evaluating fuzz testing”. In: *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2018.

[38] Karl Koscher, Tadayoshi Kohno, and David Molnar. “SURROGATES: Enabling Near-Real-Time Dynamic Analyses of Embedded Systems”. In: *9th USENIX Workshop on Offensive Technologies (WOOT ’15)*. 2015.

[39] Tamás Kovácsházy et al. “System architecture for Internet of Things with the extensive use of embedded virtualization”. In: *2013 IEEE 4th International Conference on Cognitive Infocommunications (CogInfoCom)*. 2013.

[40] Tomasz Kuchta and Bartosz Zator. “Auto Off-Target: Enabling Thorough and Scalable Testing for Complex Software Systems”. In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 2023.

[41] Sekar Kulandaivel et al. “CANNON: Reliable and Stealthy Remote Shutdown Attacks via Unaltered Automotive Microcontrollers”. In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021.

[42] Chongqing Lei et al. “A Friend’s Eye is A Good Mirror: Synthesizing MCU Peripheral Models from Peripheral Drivers”. In: *33rd USENIX Security Symposium (USENIX Security 24)*. 2024.

[43] Hao Li et al. “FEMU: A firmware-based emulation framework for SoC verification”. In: *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/-software codesign and system synthesis*. 2010.

[44] Wenqiang Li et al. “From library portability to para-rehosting: Natively executing microcontroller software on commodity hardware”. In: *arXiv preprint arXiv:2107.12867* (2021).

[45] Jay P Lim and Santosh Nagarakatte. “Automatic equivalence checking for assembly implementations of cryptography libraries”. In: *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2019.

[46] Qiang Liu et al. “FirmGuide: Boosting the Capability of Rehosting Embedded Linux Kernels through Model-Guided Kernel Execution”. In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2021.

[47] Yuwei Liu et al. “AFGen: Whole-Function Fuzzing for Applications and Libraries”. In: *2024 IEEE Symposium on Security and Privacy (SP)*. 2024.

[48] Zheyu Ma et al. “Printfuzz: Fuzzing linux drivers via automated virtual device simulation”. In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2022.

[49] Aravind Machiry et al. “DR. Checker: a soundy analysis for linux kernel drivers”. In: *26th USENIX Security Symposium (Usenix 2017)*. 2017.

[50] P.S. Magnusson et al. “Simics: A full system simulation platform”. In: *Computer* (2002).

[51] Valentin J.M. Manès et al. “The Art, Science, and Engineering of Fuzzing: A Survey”. In: *IEEE Transactions on Software Engineering* (2021).

[52] Valentin JM Manès et al. “The art, science, and engineering of fuzzing: A survey”. In: *IEEE Transactions on Software Engineering* (2019).

[53] Joel Margolis et al. “An in-depth analysis of the mirai botnet”. In: *2017 International Conference on Software Security and Assurance (ICSSA)*. 2017.

[54] Trevor Martin. *The designer’s guide to the Cortex-M processor family*. Newnes, 2016.

[55] Marius Muench et al. “Avatar 2: A multi-target orchestration platform”. In: *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)* Vol. 18. 2018.

[56] Marius Muench et al. “What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices”. In: *Network and Distributed System Security Symposium (NDSS)*. 2018.

[57] Arif Ali Mughal. “The Art of Cybersecurity: Defense in Depth Strategy for Robust Protection”. In: *International Journal of Intelligent Automation and Computing* 1.1 (2018), pp. 1–20.

[58] Aniruddhan Murali et al. “Fuzzslice: Pruning false positives in static analysis warnings through function-level fuzzing”. In: *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 2024, pp. 1–13.

[59] Eoin O’driscoll and Garret E O’donnell. “Industrial power and energy metering–a state-of-the-art review”. In: *Journal of Cleaner Production* (2013).

[60] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. “T-Fuzz: fuzzing by program transformation”. In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018.

[61] Dipika Roy Prapti et al. “Internet of Things (IoT)-based aquaculture: An overview of IoT application on water quality monitoring”. In: *Reviews in Aquaculture* (2022).

[62] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. “LOCKSMITH: context-sensitive correlation analysis for race detection”. In: *SIGPLAN Not.* (June 2006).

[63] *QEMU*. https://www.qemu.org/.

[64] *Real Time Operating System (RTOS) | Microsoft Azure — azure.microsoft.com*. https://azure.microsoft.com/en-us/products/rtos. [Accessed 07-02-2024].

[65] Edwin D Reilly. “Memory-mapped I/O”. In: *Encyclopedia of Computer Science*. 2003, pp. 1152–1152.

[66] Roman Rohleder. “Hands-on ghidra-a tutorial about the software reverse engineering framework”. In: *Proceedings of the 3rd ACM Workshop on Software Protection*. 2019.

[67] Michael Rüegg and Peter Sommerlad. “Refactoring towards seams in C++”. In: *2012 7th International Workshop on Automation of Software Test (AST)*. 2012.

[68] Jan Ruge et al. “Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets”. In: *29th USENIX Security Symposium (Usenix 20)*. 2020.

[69] *Rust4Embedded Bug Survey*. https://docs.google.com/spreadsheets/d/e/2PACX-1vQndSwy_CDJFeUCkc1PdUjF2j_q8eijUeRl8tjkM_C4D7mkGAK-QJss09j9JtIT8lSYfBNKg9-QUG7p/pubhtml.

[70] *SafeRTOS - an independently certified kernel for safety critical applications IEC61508 EN62304 and FDA 510(k) — freertos.org*. https://www.freertos.org/FreeRTOS-Plus/Safety_Critical_Certified/SafeRTOS.html. [Accessed 16-02-2024].

[71] Tobias Scharnowski et al. “Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing”. In: *31st USENIX Security Symposium (USENIX Security 22)*. 2022.

[72] Lukas Seidel, Dominik Christian Maier, and Marius Muench. “Forming Faster Firmware Fuzzers.” In: *32nd USENIX Security Symposium (USENIX 23)*. 2023.

[73] *Semiconductor Partners - FreeRTOS — freertos.org*. https://www.freertos.org/partners/semiconductor.html. [Accessed 07-02-2024].

[74] Konstantin Serebryany et al. “AddressSanitizer: A Fast Address Sanity Checker”. In: *37th USENIX Annual Technical Conference (USENIX ATC 12)*. 2012.

[75] Ayushi Sharma et al. “Rust for Embedded Systems: Current State, Challenges and Open Problems”. In: *Proceedings of the 31st ACM Conference on Computer and Communications Security (CCS)*. 2024.

[76] Ayushi Sharma et al. “Rust for embedded systems: current state, challenges and open problems”. In: *arXiv preprint arXiv:2311.05063* (2023).

[77] Mingjie Shen. “Finding 709 Defects in 258 Projects: An Experience Report on Applying CodeQL to Open-Source Embedded Software (Experience Paper) (ISSTA 2025 - Research Papers) - ISSTA 2025”. In: *ISSTA 2025* (2025).

[78] Mingjie Shen, James C. Davis, and Aravind Machiry. "Towards Automated Identification of Layering Violations in Embedded Applications (WIP)". In: *Proceedings of the 24th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. 2023.

[79] Dokyung Song et al. "SoK: Sanitizing for Security". In: *2019 IEEE Symposium on Security and Privacy (S&P)*. 2019.

[80] NB Soni and Jaideep Saraswat. "A review of IoT devices for traffic management system". In: *2017 international conference on intelligent sustainable systems (ICISS)*. 2017.

[81] Chad Spensky et al. "Conware: Automated modeling of hardware peripherals". In: *Proceedings of the 2021 ACM Asia conference on computer and communications security (AsiaCCS)*. 2021.

[82] Jayashree Srinivasan et al. "Towards rehosting embedded applications as linux applications". In: *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*. 2023.

[83] Prashast Srivastava et al. "FirmFuzz: Automated IoT Firmware Introspection and Analysis". In: *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things* (2019).

[84] John A Stankovic and R Rajkumar. "Real-Time Operating Systems". In: *Real-Time Systems* (2004).

[85] *Supported Platforms 2014; NuttX latest documentation — nuttx.apache.org*. https://nuttx.apache.org/docs/10.0.1/introduction/supported_platforms.html. [Accessed 07-02-2024].

[86] *SVD Description (*.svd) Format*. URL: https://arm-software.github.io/CMSIS%5C_5/SVD/html/svd%5C_Format%5C_pg.html.

[87] Sai Ritvik Tanksalkar et al. "LEMIX: Enabling Testing of Embedded Applications as Linux Applications (Extended Report)". In: *arXiv preprint arXiv:2503.17588* (2025).

[88] Hui Jun Tay et al. "Greenhouse: Single-Service Rehosting of Linux-Based Firmware Binaries in User-Space Emulation". In: *32nd USENIX Security Symposium (USENIX Security 23)*. 2023.

[89] timlt. *What is Microsoft Azure RTOS? — learn.microsoft.com*. https://learn.microsoft.com/en-us/azure/rtos/overview-rtos. [Accessed 16-02-2024].

[90] Fadi Al-Turjman and Joel Poncha Lemayian. "Intelligence, security, and vehicular sensor networks in internet of things (IoT)-enabled smart-cities: An overview". In: *Computers & Electrical Engineering* (2020).

[91] *URGENT/11*. en-US. https://www.armis.com/research/urgent11/. (Visited on 07/22/2023).

[92] William Von Hagen. *The definitive guide to GCC*. Apress, 2011.

[93] *VxWorks Safety Platforms — windriver.com*. https://www.windriver.com/products/vxworks/safety-platforms. [Accessed 16-02-2024].

[94] Elecia White. *Making Embedded Systems: Design Patterns for Great Software*. 2011.

[95] Christopher Wright et al. "Challenges in Firmware Re-Hosting, Emulation, and Analysis". In: *ACM Comput. Surv.* (2021).

[96] Guest Writer. *The 5 Worst Examples of IoT Hacking and Vulnerabilities in Recorded History*. en-US. https://www.iotforall.com/5-worst-iot-hacking-vulnerabilities. June 2020. (Visited on 05/01/2023).

[97] Oualid Zaazaa and Hanan El Bakkali. "Dynamic vulnerability detection approaches and tools: State of the Art". In: *2020 Fourth International Conference On Intelligent Computing in Data Sciences (ICDS)*. 2020.

[98] Jonas Zaddach et al. "AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares." In: *NDSS*. 2014.

[99] *Zephyr Project | Ecosystem Vendors — zephyrproject.org*. https://zephyrproject.org/ecosystem-vendor-offerings/. [Accessed 07-02-2024].

[100] *ZephyrRTOS*. https://zephyrproject.org/.

[101] Mingrui Zhang et al. "IntelliGen: automatic driver synthesis for fuzz testing". In: *Proceedings of the 43rd International Conference on Software Engineering: Software Engineering in Practice*. 2021.

[102] Yaowen Zheng et al. "FIRM-AFL: High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation". In: *28th USENIX Security Symposium (USENIX Security 19)*. 2019.

[103] Wei Zhou et al. "Automatic Firmware Emulation through Invalidity-guided Knowledge Inference". In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021.

A Appendix

A detailed appendix can be found in our technical report [87].

A.0.1 Examples of Continuous Scale Gradations

- **S Fidelity:** Variable-length arrays are supported by embedded GCC-based compilers but lack support in clang.
- **A Fidelity:** An emulator that implements all base instructions of ARMv7 but fails to emulate **SIMD** instructions or other processor extensions.
- **P Fidelity:** An emulator that handles GPIO and UART correctly but approximates behavior for DMA.
- **C Fidelity:** A simulation environment allows basic interrupt handling but does not handle nested interrupts.

App ID	Bug	Description	Status of Bug
f2	Assert Failure	Inconsistent use of configASSERT FreeRTOS Kernel	👤✓
f2	Assert Failure	Assert Failure in ble_event	👤🕒
f2	Build Related	Conflict of min and max from stl_algo.h in HeartRateService.h	👤✓
f3	Div By Zero	FPE in RCC_GetClocksFreq due to missing MMIO Checks	👤✓
f3	Null Deref	Null Deref in ucFlash_Write	👤🕒
f3	OOB Write	Potential undefined behavior on overlapping copy in mem_cpy	👤🕒
f4	OOB Write	Potential OOB memcpy in tud_msc_read10_cb	👤✓
f4	DoS	Infinite Loop in port_event_handle due to missing MMIO Checks	👤✗
f5	OOB Read	Potential OOB Read & Null Deref due to missing MMIO Checks in board_get_unique_id	👤🕒
n1	Build Related	Buggy handling of unsigned long in vsprintf_internal	👤✓
n3	Build Related	Compilation failure due to improper handling of CAN utils lely-core package	👤✓
n4	OOB Write	Undefined behaviour on partial overlapping copy in sim_copyfullstate	👤✓
n5	DoS	Infinite Loop in up_enable_dcache due to invalid MMIOs	👤🕒
z1	OOB Write	Stack Overflow in buf_char_out if CONFIG_PRINTK_BUFFER_SIZE is 0	👤+
z1	Null Deref	Null dereference in z_nrf_clock_control_lf_on	👤+
z2	OOB Write	The extract_conversion function in z_cbvprintf_impl can cause a potential 1 byte OOB read when the format string ends with a % character	👤+
z2	OOB Write	If bpe points to a single byte, encode_uint may cause a 1-byte underflow write by decrementing and dereferencing bp in the loop.	👤+
z2	OOB Write	Unchecked length can cause potential overflow	👤+
z3	OOB Read	OOB reads in lv_txt_utf8_next	👤✓
z3	OOB Read	Reads past the buffer possible in u8_to_dec	👤+
z3	Div By Zero	Potential div by zero by passing 0 as data frame size	👤✗
t4	Div By Zero	division by zero is possible given RCC->PLLCFGR is 0	👤✓

Table 7: Summary of all reported bugs and their statuses - 👤✓ : acknowledged and PR merged, 👤+ : acknowledged, 👤🕒 : no response (issue open), 👤✗ : not acknowledged as bug and closed.

RTOS	CATEGORY	Low Fidelity	High Fidelity
FreeRTOS	BOF	3	0
	OOB	5	0
	UAF	4	0
	Int Overflow	8	1
	Privelege Escalation	0	1
Zephyr	BOF	10	2
	OOB	5	1
	Int Overflow	3	1
	DoS	4	1
	Privelege Escalation	0	2
	NULL Deref	4	0
RIOT	BOF	2	1
	OOB	3	0
	NULL Deref	5	0
	DOS	2	1
	logic	2	0
Total		**60**	**11**

Table 8: Our analysis of the CVEs from the survey [69] conducted by Rust4Embedded (Extended Report) [76] indicates that only 11 out of 71 (15%) require high-fidelity execution (i.e precise hardware modeling).

```c
__STATIC_FORCEINLINE uint32_t __get_IPSR(void)
{
    uint32_t result;
    ⚙__ASM volatile ("MRS %0, ipsr" : "=r" (result));
    ➕ result = random() % 2; ➕
    return result;
}
#define FURI_IS_IRQ_MODE() ({__get_IPSR() != 0}) ✅

bool furi_kernel_is_irq_or_masked(void) {
    return {FURI_IS_IRQ_MODE()};}
int main(void) {
    // furi_check handles assertions
    furi_check(
       {!furi_kernel_is_irq_or_masked()} ✅
    );
    return 0;
}
```

Listing 4: An example from Flipperzero shows inline assembly (⚙) removed by LEMIX, and injected code (➕) to reinitialize with a random value. Highlighted checks (✅) verify the IPSR to ensure execution is not in IRQ context.

```c
__ALIGN(16)
static const uint16_t delay_machine_code[] = {
    // SUBS r0, #loop_cycles
    0x3800 + NRFX_COREDEP_DELAY_US_LOOP_CYCLES,
    0xd8fd, // BHI .-2
    0x4770  // BX LR
};

typedef void (* delay_func_t)(uint32_t);
const delay_func_t delay_cycles =
    // Set LSB to 1 to execute the code in the
    // Thumb mode.
    (delay_func_t)((delay_machine_code) | 1));
uint32_t cycles =
    time_us * NRFX_DELAY_CPU_FREQ_MHZ;
delay_cycles(cycles);
```

Listing 5: An example of inline assembly masked inside hexadecimal machine code in Infinitime.

```c
static void z_sys_init_run_level(
    enum init_level level
)
{
   static const struct init_entry *levels[] = {
       __init_EARLY_start,
       __init_PRE_KERNEL_1_start,
       __init_PRE_KERNEL_2_start,
       __init_POST_KERNEL_start,
       __init_APPLICATION_start,
   };
   const struct init_entry *entry;
   // The entries are function pointers that
   // are expected to be placed in memory in
   // the correct order. This ensures that
   // the comparison between the current entry
   // and the next one is valid.
   for (entry = levels[level]; entry <
        levels[level+1]; entry++) {
           dev->state->initialized = true;
            (void)entry->init_fn.sys();
       }
   }
}
```

Listing 6: Listing shows Layout Specific Execution found in one of Zephyr Kernel's initialization routines. The kernel expects function pointers to be present in adjacent memory locations as directed by Board Specific Linker Scripts.

```c
uint8_t ull_scan_rsp_set(struct ll_adv_set *adv,
uint8_t len, void *data)
{
    struct pdu_adv *pdu;
    pdu = lll_adv_scan_rsp_alloc(&adv->lll, &idx);
    /* update scan pdu fields. */
    ...
    /* len is attacker controlled */
    pdu->len = BDADDR_SIZE + len; 🕵
    /* OOB write at scan_rsp.data[0] */
    memcpy(&pdu->scan_rsp.data[0], data, len); 🐞
    ...
    return 0;
}
```

Listing 7: CVE-2021-3581: A low-fidelity vulnerability where unchecked length can cause OOB write if data and len are attacker controlled.

```asm
 /* validate syscall limit */
ldr ip, =K_SYSCALL_LIMIT
cmp r6, ip
/* The supplied syscall_id must be lower than the
  *  limit (Requires unsigned integer comparison)
  */
blt valid_syscall_id 🐞

/* bad syscall id.  Set arg1 to bad id and set
     call_id to SYSCALL_BAD */
str r6, [r0]
ldr r6, =K_SYSCALL_BAD
```

Listing 8: CVE-2020-10027: A high-fidelity vulnerability where a signed comparison in ARM syscall validation (blt vs. blo) allows privilege escalation from user thread to kernel.

```c
#ifndef CONFIG_PRINTK_BUFFER_SIZE
#define CONFIG_PRINTK_BUFFER_SIZE 0 ⚠
struct buf_out_context {
    char buf[CONFIG_PRINTK_BUFFER_SIZE];
    unsigned int buf_count;
};

static int buf_char_out(int c, void *ctx_p) {
    struct buf_out_context *ctx = ctx_p;
    ctx->buf[ctx->buf_count] = c; 🐞
    // buf_count incremented before the check
    ++ctx->buf_count; 💀
    if (ctx->buf_count == CONFIG_PRINTK_BUFFER_SIZE) {
        buf_flush(ctx);
    }
    return c;
}
```

Listing 9: OOB write in Zephyr function *buf_char_out*.