Towards Rehosting Embedded Applications as Linux Applications

Jayashree Srinivasan, Sai Ritvik Tanksalkar, Paschal C. Amusuo, James C. Davis, Aravind Machiry
Purdue University, USA
{srinivaj, stanksal, pamusuo, davisjam, amachiry}@purdue.edu

***Abstract*—Dynamic analysis of embedded firmware is a necessary capability for many security tasks, *e.g.,* vulnerability detection. Rehosting is a technique that enables dynamic analysis by facilitating the execution of firmware in a host environment decoupled from the actual hardware. Current rehosting techniques focus on high-fidelity execution of the entire firmware. Consequently, these techniques try to execute firmware in an emulated environment, with precise models of hardware (*i.e.,* peripheral) interactions. However, these techniques are hard to scale and have various drawbacks.**

We propose a novel take on rehosting by focusing on the application components and their interactions with the firmware without the need to model hardware dependencies. We achieve this by rehosting the embedded application as a Linux application. In addition to avoiding precise peripheral modeling, our rehosting technique enables the use of existing dynamic analysis techniques on these embedded applications. We provide an overview of our approach and demonstrate its feasibility on three real-world embedded applications. Our testing of these rehosted applications found 2 previously unknown defects in driver components. We discuss challenges in automating our process and present possible future research directions.

***Index Terms*—Embedded Systems, Cybersecurity, Firmware, Rehosting, Real-time Systems, RTOS, Dynamic Analysis**

I. Introduction

Embedded systems are widely used in safety-critical applications, such as medical devices [1], surveillance [2], and automotive systems [3]. These systems must be validated to ensure they deliver functionality without defects or cybersecurity vulnerabilities [4]–[6]. These systems' asynchronous and event-driven nature [7] makes it hard to apply static vulnerability detection techniques [8]. Researchers have resorted to performing dynamic analysis [9]. However, performing scalable and effective dynamic analysis has various issues [8]: need for hardware, challenges in instrumentation, input generation, and crash detection [10]. Rehosting [11] is a well-known technique to handle this. The main principle is to decouple the firmware from the hardware and execute it in an emulated environment.

Most existing techniques take a firmware targeted for a specific Microcontroller Unit (MCU) and faithfully execute it in an emulator. They model peripheral behavior using various techniques, such as manually created models [12], pattern-based model generation [13], or models built using machine learning techniques [14], [15]. They depend on the availability of an MCU-specific Instruction Set Architecture (ISA) emulator and require considerable engineering effort [16] to configure different peripherals. They focus on having a high-fidelity execution, *i.e.,* the execution of rehosted system is expected to behave nearly the same as on the corresponding hardware. Additionally, many MCUs lack supported emulators [11]. Providing emulation support to all MCUs can therefore become challenging.[1]

The authors gratefully acknowledge support from Rolls Royce.

We observe that many problems in testing embedded systems have solutions for regular applications, but that these solutions do not apply well to embedded systems. For example, several sanitizers can detect non-crash bugs but cannot be utilized in embedded systems due to the lack of required memory protections [19]. Automated input generation techniques [20], which are available for regular applications, cannot be applied to embedded systems because of their unique input channels. Additionally, some dynamic analysis methods such as Intel's PIN rely on hardware features [21], [22], and cannot be directly applied to embedded systems that lack such features.

From a cybersecurity standpoint, we argue it is better to find vulnerabilities than to delay discovery in the pursuit of perfect emulation. We, therefore, propose to rehost embedded systems as Linux applications. This will enable us to apply state-of-the-art techniques to embedded systems software. Given an embedded application, our goal is to obtain a semantically equivalent Linux executable to which inputs can be provided. There are three challenges: (1) *Retargeting to different ISAs:* Embedded applications have several MCU specific components that cannot be directly compiled for traditional ISAs because of differences in compiler toolchains and the presence of inline ISA-specific assembly code. We should have a mechanism to compile an embedded application for common desktop ISAs. (2) *Preserving Execution Semantics:* Linux applications, by default, follow single-threaded execution. However, embedded applications are engineered in terms of event-driven tasks and are multi-threaded [7]. Our rehosted Linux application should have the same execution semantics as the original embedded system, *i.e.,* task-based and event-driven execution. (3) *Handling Peripheral Interactions:* Embedded systems interact with the external world through peripheral interfaces. Peripherals are physically attached to sensors and actuators and are accessed through a special set of Memory Mapped I/O (MMIO) addresses [23]. These addresses need to

[1]We anticipate the use of general-purpose processing units to rise, especially in applications that incorporate Artificial Intelligence. However, MCUs will remain widespread. Many embedded applications require minimal computing capabilities but possess critical time constraints [17], [18].

be distinguished from regular memory accesses and handled sensibly, which is usually termed peripheral modeling.

In this paper, we present the first practical approach to rehost an embedded system as a Linux application. Further, we demonstrate its feasibility on three real-world embedded applications. We compiled these rehosted applications using sanitizers and tested them using AFL++ [24], a well-known application testing tool. We found two previously unknown bugs, indicating the effectiveness of our approach. We also provide insights into automating the retargeting and peripheral handling process, challenges, and potential solutions.

II. Background and Threat Model

Embedded systems perform a designated task with custom-designed software and hardware. Following previous systematization works [10], [11], these systems can be categorized into three types: Type-1 systems use general purpose Operating System (OS) retrofitted for embedded systems, *e.g.,* Embedded Linux; Type-2 systems use a Real Time Operating System (RTOS); and Type-3 systems use no OS abstractions. In this work, we focus on Type-2 systems. As shown in Figure 1, they have a layered design [25]. Application logic is implemented in tasks managed by an RTOS. Most RTOSes modularize their code base to capture all the hardware-specific functionalities within a portability layer specialized for each supported MCU.

These systems use peripherals to function and interact with the external world. The interaction with these peripherals is done through Memory Mapped I/O (MMIO), *i.e.,* by reading from/writing to a dedicated memory region. From a functionality perspective, peripherals can be classified into: (i) *Essential*: These are necessary for the execution of the system, like the ones pertaining to the clock module or power source. Most existing rehosting works [12]–[15] focus on modeling accesses to essential peripherals. (ii) *Non-essential*: These are not necessary for execution but provide interfaces to the external world, *e.g.,* temperature sensor connected via a GPIO.

These systems execute in an event-driven way [26]. Peripherals trigger interrupts that switch execution from a currently executing task to a service routine.

Threat Model: We assume that non-essential peripherals can receive data from malicious sources. In the case of essential peripherals, they can be malformed and provide values not expected by the application code. Furthermore, we assume that these peripherals are only connected via the system under test and do not have out-of-system interactions with each other.

In embedded systems, the data from external entities (*e.g.,* network packets) is received through non-essential peripherals, creating a possibility of memory exploits [27]. Additionally, in the case of essential peripherals (*e.g.,* clock), any deviation from the valid behavior could possibly create a scenario that is not expected (*e.g.,* disabling a safety mechanism) [28].

III. Overview of Our Approach

This section summarizes our approach to rehosting an embedded application as a Linux application. Figure 2 shows

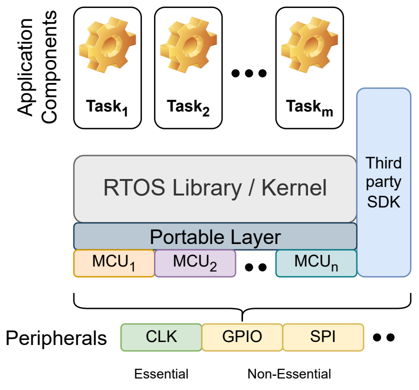

Fig. 1. Architecture of Type-2 Embedded Systems: The application tasks execute over the RTOS layer. The portable layer acts as a bridge between application tasks, MCU and third-party SDKs, providing OS specific standard APIs for hardware interactions, especially the peripherals which may be essential like the Clock or non-essential like GPIO, SPI, etc.

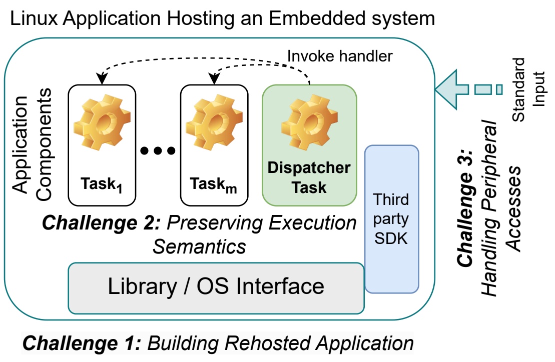
Linux Application Hosting an Embedded system
Invoke handler
Application Components
Task$_1$
Task$_m$
Dispatcher Task
Third party SDK
Standard Input
Challenge 2: Preserving Execution Semantics
Library / OS Interface
Challenge 3: Handling Peripheral Accesses
Challenge 1: Building Rehosted Application

Fig. 2. Overview of our Approach and Associated Challenges. The rehosted system consists of the application components, the RTOS, and the other SDK components. The main challenges are (1) building this rehosted application, (2) preserving the execution semantics, and (3) handling peripherals.

an embedded application rehosted as a Linux application using our approach to handling the three challenges.

A. *Retargeting for a general-purpose ISA (*x86-64*)*

As explained in Section I, to compile an application for x86-64, we need to handle: (a) Toolchain differences between MCU specific compiler and the x86-64 compiler. (b) Inline assembly. Most MCUs use a RISC-based ISA with custom toolchains suitable for the architecture of the processing unit. These support architecture-specific options (*e.g.,* `-mthumb-interwork`) and custom options (*e.g.,* `-grecord-gcc-switches`), often unavailable in commodity compilers (*e.g.,* Clang). Consequently, naively replacing the compiler with a commodity compiler fails. Using existing build interceptor tools [29], we replace the embedded toolchain and the dependent flags relevant to compilation and linking, with Linux equivalents in the Clang toolchain.

The presence of inline assembly code pertaining to the MCU architecture is a related challenge. We noticed that inline assembly is mostly used to access MCU specific registers and modify features of the MCU. As our main focus is on

```
void USART1_IRQHandler(void)
{
   // Check if USART Receive Status is Set
   if(USART_GetITStatus(USART1, USART_IT_RXNE)==SET)
   {
       // Reveive data from USART1 Data Registers
       cChar = (uint8_t)USART_ReceiveData(USART1);
        ...
   } //...
}
```

Listing 1: Example Interrupt Handler: The handler consists of highly constrained checks on the register values before trying to read from the USART (peripheral).

vulnerability detection, our intuition is that NOP'ing out inline assembly components would not overly hinder the overall analysis. We therefore handle inline assembly by commenting it out from the source files.

B. Preserving Execution Semantics

The second challenge is to ensure that the application components (*e.g.,* tasks) in a rehosted Linux application have the same execution semantics as in the target embedded system.

Task-based Execution: We noticed that many popular RTOSes have a Linux port [30]–[32], which implements the portable layer (Figure 1) using Linux-based user space libraries, *e.g.,* providing task interface through `pthreads` library APIs. Consequently, we can have the RTOS compiled for Linux, where all its functions (*e.g.,* task, message queues) are implemented through Linux user space libraries. We use these ports by compiling the embedded application using the Linux portable layer instead of the MCU alternative. The resulting executable will run as a Linux application — using existing library APIs (*e.g.,* `pthread`) to achieve nearly the same execution semantics as in the original embedded system.

Event-driven Execution: Interrupts enable event-driven execution by invoking handler functions (Listing 1) when triggered. These handlers are recorded in the interrupt vector table which is usually present in a MCU vendor-provided assembly file (*eg., startup.a*). This cannot be executed on the x86 machine. The application can register custom interrupts by adding the handler addresses to the interrupt vector table. We recognize such interrupt-registering functions in the application and gather all the handlers. We then create a special dispatcher task (Figure 2) which will be invoked periodically by the RTOS scheduler. The interrupt handlers will be executed as function calls by this task as shown below:

```
handlers <- array of registered handlers.
n <- number of handlers.

a = read_input();
idx = a % n;
handlers[idx](); // invoke a handler.
```

C. Handling Peripheral Interactions

Peripherals are accessed through special memory regions, called MMIO regions, which need to be identified and handled.

Identifying MMIO regions: First, we need a way to distinguish MMIO access from normal memory access. We observed that MMIO address ranges are usually hardcoded in specific source files (*e.g.,stm32f4xx.h* [33]). Therefore, we consider these addresses, usually within a specific range that is indicated in the MCU datasheet as MMIO regions.

Peripheral Access Handling: We analyze source files of a given application to identify accesses to MMIO regions. Our threat model supposes that peripherals may be producing malicious input. We, therefore, modify these accesses to use standard input, so that a dynamic analysis can observe the effect of different input values. Specifically, as we show below, all reads from these regions will be replaced with reads from standard input (❮), and all writes will be ignored — this is in line with our threat model (Section II).

```
// ...
// Reading values from hardware registers
// pllm = RCC->PLLCFGR & RCC_PLLCFGR_PLLM;
pllm = ❮read_standard_input() & RCC_PLLCFGR_PLLM;
```

The Linux port handles the execution semantics without depending on essential peripherals. Hence feeding random data to these doesn't affect the execution semantics of the rehosted application. Feeding data from standard input to non-essential peripheral accesses, such as GPIO and SPI, models the embedded application's external world interactions through standard input of the corresponding rehosted application.

Handling Peripheral Access Checks: Peripherals use control registers to communicate the availability of data. Applications check for specific values in control registers before trying to read data (Listing 1). We noticed that most of these conditions are non-data-dependent, meaning that the code within the conditional expression does not use the value in the condition. We handle this by making such non-data-dependent MMIO access-based conditions always true. This approach follows the previous work T-fuzz [34], which showed that ignoring non-data-dependent conditions improves code coverage.

IV. Feasibility Study

We assessed the feasibility of our approach by applying it to three real-world applications based on FreeRTOS [35].

InfiniTime: A smart watch application [36] based on nrf52 microcontroller. It uses many libraries — LVGL for UI, NimBLE for its BlueTooth Stack, and JetBrains for fonts.

TinyUSB demo: A demo application of TinyUSB [37], a popular cross-platform USB Host/Device stack for embedded systems that runs on feather_nrf52840_express board.

SmartSpeaker: A smart speaker application [38] using stm32f407vet6, wm8978, and esp8266 for master control, audio DA/ADC, and network communications, respectively.

We converted each of the above applications into Linux applications using our approach (Section III). Specifically, we built the applications using the Linux port of FreeRTOS, by using compiler flags with that of x86-64 Clang. We then manually identified all MMIO accesses and replaced them with reads from standard input. Finally, we created our interrupt dispatcher task that invokes one of the registered interrupt handlers based on a value read from standard input.

Preliminary Evaluation: We compiled each of these applications using ASAN [39] and executed them on Ubuntu 20.04 machine. The application tasks executed as expected without

```
1   __weak uint32_t HAL_RCC_GetSysClockFreq(void)
2   {
3     //...
4     // Reading values from hardware registers
5     pllm = RCC->PLLCFGR & RCC_PLLCFGR_PLLM;
6     if(__HAL_RCC_GET_PLL_OSCSOURCE()
7        != RCC_PLLSOURCE_HSI)
8     {
9       // Usage of read value as divisor without checks
10      pllvco = (uint32_t) ((((uint64_t) HSE_VALUE *
11      ((uint64_t) ((RCC->PLLCFGR & RCC_PLLCFGR_PLLN)
12      >> RCC_PLLCFGR_PLLN_Pos)))) / (uint64_t)pllm 🐞);
13    } //...
14  }
```

Listing 2: Division By Zero Error: Value *pllm* is read from a peripheral register and used as divisor without any check (🐞).

undefined crashes. Accesses to peripheral regions did not fault. Execution proceeded as input was provided via standard input.

Testing: To check the robustness and bug-exposing ability of these rehosted applications, we tested them using AFL++ [40] for 24 hours. We found two previously unknown floating-point exceptions in the STM32 driver code [33] — in the Clock module and in the SPI driver. As seen in Listing 2, the value `pllm` is read from a member of `RCC` peripheral register (line 5). On line 12, it is used as a divisor without any check for zero. STM32 has fixed the bug [41] in their latest release.

V. Discussion and Future Work

The scalability of our approach lies in the fact that most of it can be automated and generalized, which we discuss here.

Automated Retargeting: Manual intervention for retargeting steps like commenting out inline assembly, and handling compiler errors because of inconsistent code practices (invalid pointer casts) can be automated via source-to-source transformation. A recent tool, 3c [42], demonstrated the feasibility of such an approach by modifying C code to add type annotations automatically. For certain errors, such as linking with undefined references to a symbol, it is harder to provide a completely automatic approach. Therefore, we plan a human-in-the-loop approach where our technique guides an engineer.

Automated Peripheral Handling: Handling peripherals automatically involves two tasks: (1) Identifying MMIO address ranges; (2) Modifying these accesses with read from standard input. The first can be automated using static analysis to identify hardcoded addresses. However, handling the second task just by using static analysis is hard as pointers to MMIO addresses can be passed as arguments to functions. Consequently, we need precise pointer analysis – a known hard problem [43] – to identify which pointer accesses to be modified. We plan to use a dynamic checking approach. We can statically instrument all reads/writes; if a read is from an MMIO address, then we return data from standard input and ignore writes to MMIO addresses. Also, we plan to engineer a configurable and feedback-driven mechanism for triggering interrupts instead of a dispatcher task that is currently used.

Effective Testing: As explained in Section III-C, embedded applications have non-data-dependent checks to read data from peripherals (*i.e.,* input data). Currently, we handle this by manually disabling such checks. However, this identification can be automated by using data dependency analysis [44] and disabling them through compiler instrumentation. We also believe that embedded applications could be a good candidate for directly fuzzing individual functions owing to less complex functions that require structured data, especially at the hardware-interface level (Listing 2). We plan to explore the function-level fuzzing of the rehosted applications.

Extension to other RTOS: While we considered applications using FreeRTOS [45], our approach can be extended to other open-source RTOSes like Zephyr [46] and Nuttx [47] as they have the Linux port required by our retargeting scheme.

Systematic Evaluation: We plan to do a systematic evaluation of the engineering cost vs. fidelity tradeoff, with application level analysis in terms of coverage and vulnerability detection.

VI. Related Work

Rehosting [11] has been an active area of research for the past decade. Rehosting techniques can be categorized into high-level or OS emulation, hardware-assisted partial emulation, peripheral emulation through symbolic execution, machine learning, or a combination of these.

Complete high-level emulation targets OS-level abstractions. Although this works well for Type-1 systems [48], [49], which use standard OSes, these cannot be applied to Type-2 systems because of the lack of a well-defined OS interface. Consequently, no work exists that can perform high-level emulation using RTOS on Type-2 systems [11]. The rest of the categories focus on handling peripheral access by executing an embedded system in an emulator. Hardware-assisted partial emulation techniques use real hardware to handle peripheral accesses. Few works employ manually or semi-automatically created models for peripherals [12], [13], [50]. ML-based techniques [14], [15] first record the pattern of peripheral accesses and then try to use ML models to create access patterns for these peripherals. Some techniques [51], [52] use symbolic execution [53] to create peripheral models.

As shown by the recent systematization work [11], the existing techniques are hard to extend for different peripherals and depend on the existence of emulators [54], [55] of the corresponding ISA. We are the first to bring a new dimension to rehosting by exploring the possibility of rehosting embedded application components (instead of the entire system) by converting them to Linux applications. As shown by our preliminary evaluation, our approach is feasible and can help in finding hard-to-trigger bugs in embedded system components.

VII. Conclusion

We propose a new approach to rehosting embedded applications as Linux applications by providing solutions to the associated challenges of retargeting to x86-64, preserving the execution semantics, and handling the peripheral interactions. We demonstrated the feasibility of our approach through a preliminary study of three real-world applications. Our testing of these rehosted Linux applications found two new bugs in heavily used components. We also present possible techniques and future plans to automate and extend our approach.

References

[1] N. Arandia, J. I. Garate, and J. Mabe, "Embedded sensor systems in medical devices: Requisites and challenges ahead," *Sensors (Basel)*, vol. 22, no. 24, p. 9917, Dec. 2022.

[2] A. Goel, C. Tung, X. Hu, H. Wang, J. C. Davis, G. K. Thiruvathukal, and Y.-H. Lu, "Low-power multi-camera object re-identification using hierarchical neural networks," in *2021 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 2021, pp. 1–6.

[3] J. Garcia, Y. Feng, J. Shen, S. Almanee, Y. Xia, and Q. A. Chen, "A comprehensive study of autonomous vehicle bugs," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 385–396. [Online]. Available: https://doi.org/10.1145/3377811.3380397

[4] D. Papp, Z. Ma, and L. Buttyan, "Embedded systems security: Threats, vulnerabilities, and attack taxonomy," in *2015 13th Annual Conference on Privacy, Security and Trust (PST)*, 2015, pp. 145–152.

[5] E. White, *Making embedded systems: Design Patterns for Great Software*. O'Reilly, 2012.

[6] D. Anandayuvaraj and J. C. Davis, "Reflecting on recurring failures in iot development," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–5.

[7] T. A. Henzinger and J. 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*. Springer, 2006, pp. 1–15.

[8] A. Qasem, P. Shirani, M. Debbabi, L. Wang, B. Lebel, and B. L. Agba, "Automatic vulnerability detection in embedded devices and firmware: Survey and layered taxonomies," *ACM Comput. Surv.*, vol. 54, no. 2, mar 2021. [Online]. Available: https://doi.org/10.1145/3432893

[9] J. Yun, F. Rustamov, J. Kim, and Y. Shin, "Fuzzing of embedded systems: A survey," *ACM Comput. Surv.*, vol. 55, no. 7, dec 2022. [Online]. Available: https://doi.org/10.1145/3538644

[10] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices," in *Network and Distributed System Security Symposium (NDSS)*, 2018.

[11] A. Fasano, T. Ballo, M. Muench, T. Leek, A. Bulekov, B. Dolan-Gavitt, M. Egele, A. Francillon, L. Lu, N. Gregory *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, pp. 687–701.

[12] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "HALucinator: firmware re-hosting through abstraction layer emulation," in *Proceedings of the 29th USENIX Conference on Security Symposium*, ser. SEC'20. USA: USENIX Association, Aug. 2020, pp. 1201–1218.

[13] B. Feng, A. Mera, and L. Lu, "P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1237–1254. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/feng

[14] C. Spensky, A. Machiry, N. Redini, C. Unger, G. Foster, E. Blasband, H. Okhravi, C. Kruegel, and G. Vigna, "Conware: Automated modeling of hardware peripherals," in *Proceedings of the 2021 ACM Asia conference on computer and communications security (AsiaCCS)*, 2021, pp. 95–109.

[15] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel *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, pp. 135–150.

[16] C. Wright, W. A. Moeglein, S. Bagchi, M. Kulkarni, and A. A. Clements, "Challenges in firmware re-hosting, emulation, and analysis," *ACM Comput. Surv.*, vol. 54, no. 1, jan 2021. [Online]. Available: https://doi.org/10.1145/3423167

[17] R. Chéour, S. Khriji, M. abid, and O. Kanoun, "Microcontrollers for iot: Optimizations, computing paradigms, and future directions," in *2020 IEEE 6th World Forum on Internet of Things (WF-IoT)*, 2020, pp. 1–7.

[18] F. Vahid and T. Givargis, *Embedded System Design: A Unified Hardware/Software Introduction*, 1st ed. USA: John Wiley & Sons, Inc., 2001.

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

[20] M. Böhme, C. Cadar, and A. Roychoudhury, "Fuzzing: Challenges and reflections." *IEEE Software*, vol. 38, no. 3, pp. 79–86, 2021.

[21] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 190–200. [Online]. Available: https://doi.org/10.1145/1065010.1065034

[22] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "libdft: Practical dynamic data flow tracking for commodity systems," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, 2012, pp. 121–132.

[23] E. D. Reilly, "Memory-mapped i/o," in *Encyclopedia of Computer Science*, 2003, pp. 1152–1152.

[24] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++ combining incremental steps of fuzzing research," in *Proceedings of the 14th USENIX Conference on Offensive Technologies*, 2020, pp. 10–10.

[25] M. Shen, J. C. Davis, and A. Machiry, "Towards automated identification of layering violations in embedded applications (wip)," in *2023 ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*. ACM, 2023.

[26] J. Davis, A. Thekumparampil, and D. Lee, "Node. fz: Fuzzing the server-side event-driven architecture," in *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*, 2017, pp. 145–160.

[27] M. A. Arroyo, "Bespoke security for resource constrained cyber-physical systems," in *ProQuest Dissertations and Theses*. Columbia University, 2021, p. 171, accessed 15 Feb. 2023. [Online]. Available: https://www.proquest.com/dissertations-theses/bespoke-security-resource-constrained-cyber/docview/2470276679/se-2?accountid=13360.

[28] S. Kulandaivel, S. Jain, J. Guajardo, and V. Sekar, "Cannon: Reliable and stealthy remote shutdown attacks via unaltered automotive microcontrollers," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 195–210.

[29] travitch, "Whole Program LLVM," https://github.com/travitch/whole-program-llvm, 2015.

[30] osrtos, "List of open source real-time operating systems," https://www.osrtos.com/, 2023.

[31] M. Silva, D. Cerdeira, S. Pinto, and T. Gomes, "Operating systems for internet of things low-end devices: Analysis and benchmarking," *IEEE Internet of Things Journal*, vol. 6, no. 6, pp. 10 375–10 383, 2019.

[32] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes, "Operating systems for low-end devices in the internet of things: A survey," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 720–734, 2016.

[33] STMElectronics, "Stm32Lib," https://github.com/STMicroelectronics/STM32CubeF4.

[34] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-fuzz: fuzzing by program transformation," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 697–710.

[35] "FreeRTOS," http://freertos.org.

[36] InfinitimeOrg, "Infinitime," https://github.com/InfiniTimeOrg/InfiniTime.

[37] hathach, "TinyUSB," https://github.com/hathach/tinyusb.

[38] lovelyterry, "SmartSpeaker," https://github.com/lovelyterry/SmartSpeaker.

[39] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, Jun. 2012, pp. 309–318. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[40] M. Zalewski, "Afl technical details," 2018. [Online]. Available: https://lcamtuf.coredump.cx/afl/technical_details.txt

[41] STMElectronics, "Stm32Lib Bug," https://github.com/STMicroelectronics/STM32CubeF4/pull/154.

[42] A. Machiry, J. Kastner, M. McCutchen, A. Eline, K. Headley, and M. Hicks, "C to checked c by 3c," *Proceedings of the ACM on Programming Languages*, vol. 6, no. OOPSLA1, pp. 1–29, 2022.

[43] M. Hind, “Pointer analysis: Haven’t we solved this problem yet?” in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2001, pp. 54–61.

[44] R. Bisbey, J. Carlstedt, D. Chase, D. Hollingworth *et al.*, “Data dependency analysis.” University of Southern California, Marina del Rey Information Sciences Institute, Tech. Rep., 1976.

[45] F. Guan, L. Peng, L. Perneel, and M. Timmerman, “Open source freertos as a case study in real-time operating system evolution,” *Journal of Systems and Software (JSS)*, vol. 118, pp. 19–35, 2016.

[46] “ZephyrRTOS,” https://zephyrproject.org/.

[47] “NuttxRTOS,” https://nuttx.apache.org/.

[48] D. D. Chen, M. Woo, D. Brumley, and M. Egele, “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. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/towards-automated-dynamic-analysis-linux-based-embedded-firmware.pdf

[49] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, “FIRM-AFL: High-Throughput greybox fuzzing of IoT firmware via augmented process emulation,” in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1099–1114. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/zheng

[50] W. Zhou, L. Guan, P. Liu, and Y. Zhang, “Automatic firmware emulation through invalidity-guided knowledge inference,” in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/zhou

[51] T. Scharnowski, N. Bars, M. Schloegel, E. Gustafson, M. Muench, G. Vigna, C. Kruegel, T. Holz, and A. Abbasi, “Fuzzware: Using precise MMIO modeling for effective firmware fuzzing,” in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 1239–1256. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/scharnowski

[52] N. Corteggiani, G. Camurati, and A. Francillon, “Inception: System-Wide security testing of Real-World embedded systems software,” in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 309–326. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/corteggiani

[53] “KLEE.” [Online]. Available: http://klee.github.io/

[54] “QEMU.” [Online]. Available: https://www.qemu.org/

[55] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, “Simics: A full system simulation platform,” *Computer*, vol. 35, no. 2, pp. 50–58, 2002.