Towards Automated Identification of Layering Violations in Embedded Applications (WIP)

Mingjie Shen
shen497@purdue.edu
Purdue University
West Lafayette, IN, USA

James C. Davis
davisjam@purdue.edu
Purdue University
West Lafayette, IN, USA

Aravind Machiry
amachiry@purdue.edu
Purdue University
West Lafayette, IN, USA

Abstract

For portability, embedded systems software follows a layered design to reduce dependence on particular hardware behavior. We consider the problem of identifying layering violations: instances where the embedded application accesses non-adjacent layers. This paper presents our preliminary work to detect a class of layering violations called *Non Conventional MMIO Accesses (NCMAs)*. We find them by searching for direct Memory Mapped Input Output (MMIO) accesses made outside of the Hardware Abstraction Layer (HAL). For evaluation, we curated a list of 988 applications spanning 5 Real Time Operating Systems (RTOSes) – *the first large dataset of compilable embedded applications.* Our system identified 369 NCMAs. We reported these issues to the corresponding developers and found interesting reasons for committing layering violations. We have open-sourced our tool and the collected dataset to foster future research.

CCS Concepts: • **Computer systems organization** → **Firmware**; **Embedded software**.

Keywords: Embedded Systems, Portability, Firmware, Hardware Abstraction Layer, Static Analysis

ACM Reference Format:
Mingjie Shen, James C. Davis, and Aravind Machiry. 2023. 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 (LCTES '23), June 18, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3589610.3596271

1 Introduction

Embedded software (*i.e.,* "IoT/cyber-physical") enables critical systems [12, 24]. As explained by the recent work [14], embedded systems can be classified into three categories based on their software architecture. This work focuses on deeply embedded or Type-2 systems, whose ecosystem has a lot of diversity in terms of hardware [14, 28] (Micro Controller Unit (MCU) and peripherals) and supported software. For instance, there are 31 different Real Time Operating Systems (RTOSes) [1].

LCTES '23, June 18, 2023, Orlando, FL, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0174-0/23/06.
https://doi.org/10.1145/3589610.3596271

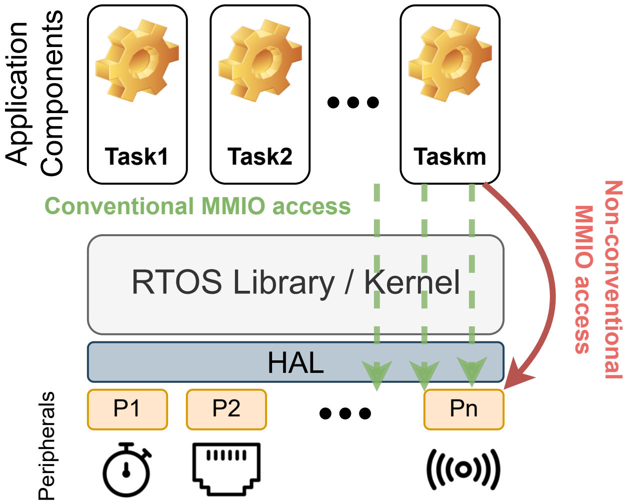

Figure 1. Software design of Type-2 Embedded Systems. Application tasks execute on a RTOS, which uses HAL to access peripherals. Conventional MMIO access (green) uses HAL, while NCMAs (red) bypass HAL via direct read/write to MMIO addresses.

The time required to port an embedded application to a new MCU or peripheral affects system cost – a major factor in embedded systems engineering [10]. To reduce the difficulty of porting, Figure 1 illustrates the typical layered design: the embedded software application is organized into layers that communicate only with adjacent layers [8]. If applications follow this design, it simplifies supporting a new MCU, as only one layer is impacted. When applications commit *layering violations*, portability degrades.

In this paper, we present preliminary work toward automatically detecting layering violations in embedded software. Specifically, we focus on a layering violation related to hardware or peripheral access, called a *Non Conventional MMIO Access (NCMA)*. Most peripherals are accessed by Memory Mapped Input Output (MMIO) [17]: they are addressed by the same address space as the main memory and are accessed via regular load/store instructions. An application *should* access these memory regions through a Hardware Abstraction Layer (HAL), *e.g.,* exposed through SDK functions. Figure 1 depicts conventional access and NCMA layering violations.

We designed a static analysis tool to automatically detect NCMAs in embedded applications. According to the layered design, the only accesses to MMIO regions should come from functions in or below the HAL. We use call graph analysis to identify functions in the HAL. We then find MMIO accesses made outside of the HAL functions – these are possible NCMAs.

To evaluate the approach, we collected 988 applications spanning 5 RTOSes – *the first large dataset of compilable embedded applications.* In this dataset, we identified 369 NCMAs. We reported these issues to maintainers and found two different rationales for NCMAs: lack of concern for portability, and workarounds for undesirable behaviors of HAL APIs.

2 Background and Motivation

Listing 1 shows an LED application in Zephyr RTOS that uses GPIO API `gpio_pin_toggle_dt(&led)` (a HAL function) to toggle an LED pin. This application compiles and runs on all boards supported by Zephyr [31] without modifying the application. Zephyr achieves this through abstraction layers (*e.g.,* peripheral APIs), which rely on the vendors' HAL functions. Figure 2 shows the call graph of the application in Listing 1, showing Zephyr abstractions for vendor-specific HALs.

On the other hand, in `ncma_example()` (also in Listing 1), there is direct access (💣) to the timer peripheral through an MMIO address – by first casting it to a struct and accessing its member. The application directly accesses peripherals through hardcoded MMIO addresses (⚠), violating the principle of layered design.

3 HalVD: Automated Detection of NCMAs

Given an embedded application, our goal is to automatically identify all NCMAs and provide source-level reports, *e.g., "There is an NCMA in function X at line Y in file Z".*

Figure 1 shows the high-level idea of conventional MMIO accesses and NCMAs. Given the application's source code, we automatically convert it to LLVM bitcode using a custom build monitoring tool. Next, we identify all MMIO accesses by performing a lightweight analysis of the LLVM IR. Finally, we use heuristics on the call graph of the application to determine whether a given MMIO access is a NCMA.

Bitcode Generation via Build Monitoring: We want to use the LLVM compiler infrastructure to implement our analysis. However, it requires bitcode of the source program. Generating whole program bitcode (one bitcode file for the entire embedded application) is challenging as it requires modifying build scripts and dealing with toolchain-specific aspects. Prior work handled this manually [7] – we (partially) automate it.

```
static const struct gpio_dt_spec led =
↪  GPIO_DT_SPEC_GET(LED0_NODE, gpios);
int main() {
  int ret; ...
  while (1) {
    ret = gpio_pin_toggle_dt(&led);
    if (ret < 0) return;
    k_msleep(SLEEP_TIME_MS);
  }
}
/* --------------------------------------------- */
typedef struct {... __IOM uint32_t CC[6]; } NRF_TIMER_Type;
#define NRF_TIMER2  ((NRF_TIMER_Type*) 0x4000A000UL ⚠)
void ncma_example() {
  SetVersion(💣NRF_TIMER2->CC[0]);
}
```

Listing 1. An LED blinker application `main()` that follows a layered design – interacts with only the adjacent layer through the function `gpio_pin_toggle_dt(&led)` (highlighted) to periodically toggle an LED pin. And an example of NCMA (💣) `ncma_example()`: `NRF_TIMER2->CC[0]` is a struct member access through the pointer `NRF_TIMER2_BASE(=0x4000A000UL)`.

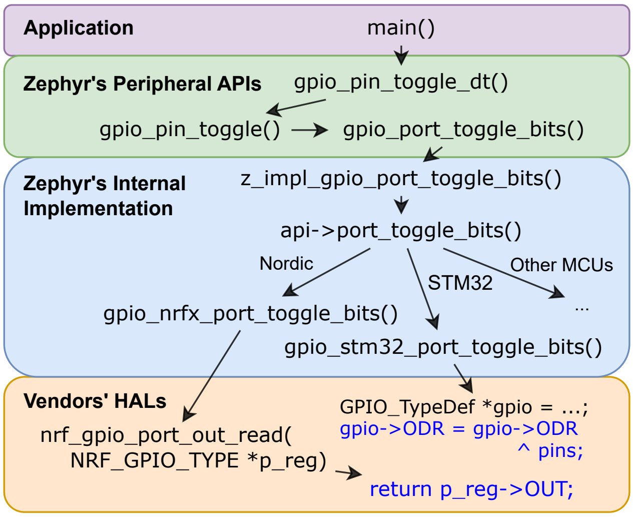

Figure 2. Call graph demonstrating the layered design of the application in Listing 1. Each layer exposes functions to the above layer and uses functions from the below layer.

We obtain whole program bitcode by using runtime build monitoring based on WLLVM [27]. Specifically, we monitor the build process of a given embedded application to capture all invocations of the compiler. We then translate each compilation command (*i.e.,* object file generation), into the corresponding bitcode generation command in CLANG. Finally, all the generated bitcode files are linked together into the whole program bitcode.

Almost all embedded systems we examined used a GCC-based toolchain, and the translation to CLANG involves four

```
define ... @ncma_example() {
  %1 = load volatile i32, i32* inttoptr (i32 1073784128 to
↪  i32*)
  tail call void @_ZN...10SetVersionEj(i32 noundef %1)
}
```

Listing 2. LLVM bitcode of `ncma_example()`. %1 corresponds to struct member access `NRF_TIMER2->CC[0]`. This `load` instruction's pointer operand is the constant `1073784128` (`0x4000A540`).

challenges. (1) Certain non-standard C/C++ features, such as variable-length struct members, are supported by GCC but not by CLANG. We manually comment out this type of code. (2) Several GCC-specific compilation options (*e.g.,* `-mfp16-format=`) are not supported by CLANG. We automatically remove them. (3) Several GCC-specific link options (*e.g.,* `-spec=`) are not supported by CLANG. We use GCC instead of CLANG for linking. (4) Various other toolchain-specific aspects must be handled, *e.g.,* passing GCC's standard system directories to CLANG. All of these treatments are semantics-preserving.

In addition to being scalable, our technique is build-system agnostic – it works for current or future build systems.

MMIO Access Finder via Constant Memory Addr: Previous work [22] shows that embedded systems access MMIO addresses through constant ("hardcoded") values; peripherals document the specific memory ranges with which they will interact. In contrast, normal variables (global or local) are rarely accessed via hardcoded addresses. For example, Listing 2 shows LLVM bitcode of function `ncma_example()`. The `load` instruction for MMIO access uses a constant address.

We design our technique based on this heuristic. Specifically, we perform static analysis on the bitcode of the firmware and identify any memory access instruction, *i.e.,* `load`, `store`, `getelementptr`, whose pointer operand is a constant. We consider such memory access instructions as MMIO accesses and record the corresponding function. The set of all functions identified in this step is denoted M_f.

NCMA Identification via Call Graph Analysis: NCMAs occur in the *non-HAL* functions that directly access MMIO addresses ($\subseteq M_f$).

We observe that HAL functions expose necessary functionality and consequently are invoked by many functions. Equivalently, in the call graph of the firmware, HAL functions are reachable from many functions. A suitable measure for this is the *transitive in-degree* (ID_t) of nodes in the call graph. Figure 3 illustrates. HAL functions (green nodes) have higher transitive in-degree than the other nodes. A parameter $\lambda < ID_t$ may distinguish HAL from non-HAL functions.

Given the call graph of the application, we compute ID_t of each node (*i.e.,* function) and apply λ to identify the non-HAL subset. We note that call graphs are large (*e.g.,* InfiniTime firmware has 7,688 functions) and an accurate $\Theta(|V|^3)$ time algorithm is impractically slow. We use a $O(|E|)$ time randomized algorithm [5, 6] that estimates ID_t of each node, where $|V|$ and $|E|$ are the number of vertices and edges in the graph, respectively. We flag as NCMA any MMIO access within the non-HAL subset.

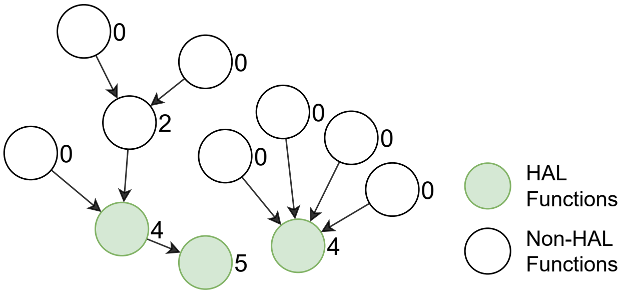

Figure 3. A hypothetical example call graph. Each node represents a function, while each edge points from caller to callee. The number near each node shows its ID_t. Assuming nodes with ID_t >= 4 are designated as HAL functions, this example has three HAL functions (in green).

4 Evaluation

We evaluated HalVD on embedded applications to assess its effectiveness and performance in detecting NCMAs.

Dataset: We collected 988 applications running on five different RTOSes: FreeRTOS [2], Zephyr [26], Mbed OS [3], Phoenix-RTOS [15], and NuttX [25]. These are real-world firmware for smartwatches, keyboards, drones, example applications provided by developers of RTOSes, and ports of well-known open-source tools. To our knowledge, this is the first large dataset of compilable real-world embedded applications. The first three columns of Table 1 summarizes the embedded applications in our dataset.

We ran HalVD on the dataset to measure its effectiveness in identifying NCMAs. It took < 3 seconds per application.

Effectiveness of MMIO Access Finder: Some functions with MMIO accesses detected by HalVD are false positives caused by HAL functions defined as C/C++ macros. There was also a case where the hard-coded address was not an MMIO address. To find if there are any false negatives, we did a random sampling of a few functions (approximately 50) where HalVD did not find any MMIO accesses. We did find false negatives, but they are expected and none of them are NCMAs, exerting no negative influence on our final goal.

Effectiveness of NCMA Identification: Table 1 shows the results of this experiment with λ set to 10. One author manually evaluated the identified NCMAs. Our call-graph based approach (Section 3) resulted in quite a few false positives and false negatives. False positives arise from HAL functions that with ID_t less than λ. Certain HAL functions which expose an uncommon peripheral may not be needed by many functions and consequently have lower ID_t. False negatives result from application functions with ID_t higher than λ. Some application functions may be needed by many

Table 1. Results of HalVD in finding NCMAs. Here, TP, FP, FN, and TN show the number of true positives, false positives, false negatives, and true negatives NCMAs, respectively. Functions calling HAL macros and other false positives of MMIO Access Finder are manually excluded before the evaluation of NCMA identification. # of NCMA Funcs = TP + FN. # of MMIO Funcs = TP + FP + FN + TN. KSLOC counts application code only, *i.e.,* excludes the code of RTOSes, SDKs, and third-party libraries.

RTOS	**Application Name** or **Category**	**# of Apps**	**# of Funcs Calling HAL Macros**	**TP**	**FP**	**FN**	**TN**	**# of NCMA Funcs**	**# of MMIO Funcs**	**KSLOC**
FreeRTOS	InfiniTime	1	21	2	17	1	127	3	147	29
	ESP-IDF examples	417	0	0	17	0	82	0	99	102
	MediaTek LinkIt examples	89	20	219	63	128	435	347	845	728
	RP2040 applications	4	1	6	20	0	52	6	78	1
	nRF52 keyboard firmware	1	0	0	4	2	58	2	64	21
	Others (5 repos)	16	114	3	98	1	25	4	127	10
Zephyr	Official examples	193	0	5	17	0	351	5	373	86
Phoenix-RTOS	Ports of open-source tools	21	0	0	6	0	0	0	6	1,240
Mbed OS	Official examples	35	0	1	11	1	129	2	142	8
NuttX	Official examples	211	0	0	208	0	1,292	0	1,500	269
Total		*988*	*156*	*236*	*461*	*133*	*2,551*	*369*	*3,381*	*2,494*

other application functions. We plan to improve this implementation in our future work (Section 5).

Developers Response: We reported some of the NCMAs found by HalVD to the developers and received a few responses. A few developers agreed that NCMAs needs to be fixed. But, there were two cases where developers believe NCMAs were fine. (i) Certain applications support only one hardware platform. Hence, non-portability resulting from NCMAs is currently acceptable; (ii) Certain HAL APIs provided by SDKs may have undesirable behaviors. For example, a timer driver might clear a register before developers can read its value. This forces developers to use NCMAs for the desired functionality. HAL developers might use this information to identify flaws in, and improve, their APIs.

A by-product of our work is that HalVD found a bug in the Bluetooth driver of Zephyr. A null pointer was used to access struct members. Since the address zero is a constant (hard-coded) address, this access is identified as an MMIO access by HalVD. We fixed the bug and opened a pull request [19], which has been merged into Zephyr's main branch.

5 Future Work

We plan work in several directions:

- **Formalizing layering violations:** We plan to create a formal definition of layering violations and prove properties related to portability.
- **Developer Studies:** We plan a developer study to understand the reasons for layering violations – this will inform the design of embedded systems and SDKs.
- **Implementation improvements:** Our current call graph-based HAL function identification has false positives and negatives. We plan to explore techniques based on the directory structure of embedded applications to identify HAL functions. We will extend the framework to identify software components other than HAL and detect layering violations, enabling developers to organize their code better.

6 Related Work

Jahnke *et al.* [11] presented a semi-automatic approach to inspect Java source code to check for violations of hardware restrictions. Schreiner *et al.* [18] compared methods of recognizing software components in embedded systems, which failed in the presence of layering violations. Martins Gomes *et al.* [13] studied the portability of several IoT operating systems by manual code review. SEAPORT [30] automatically assesses the portability of serverless applications.Existing work on detecting layering violations [9, 16, 29] cannot find NCMAs. To the best of our knowledge, no previous work proposed an automatic tool for detecting violations of HAL design in embedded applications.

Following layered design, in addition to making applications portable, also enables other retroactive security techniques [4, 21, 23].

7 Conclusion

We proposed a static analysis tool HalVD that detects a class of layering violations – NCMAs in embedded applications. We built a dataset of compilable firmware containing 988 applications and found 369 cases of NCMAs. HalVD, the dataset, and our wllvm are available at https://github.com/RTOSExploration/lctes2023-artifact and [20].

Acknowledgments

This work was partially funded by Rolls Royce under grant number 40004429.

References

[1] [n. d.]. OSRTOS. https://www.osrtos.com/

[2] Amazon Web Services, Inc. or its affiliates. 2023. FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions. https://www.freertos.org/index.html

[3] Arm Limited (or its affiliates). 2023. Mbed OS. https://os.mbed.com/mbed-os/

[4] Abraham A. Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. 2020. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1201–1218. https://www.usenix.org/conference/usenixsecurity20/presentation/clements

[5] Edith Cohen. 1994. Estimating the Size of the Transitive Closure in Linear Time. In *Proceedings 35th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 190–200.

[6] Edith Cohen. 1997. Size-Estimation Framework with Applications to Transitive Closure and Reachability. *J. Comput. System Sci.* 55, 3 (Dec. 1997), 441–453. https://doi.org/10.1006/jcss.1997.1534

[7] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. 2018. Inception: System-Wide Security Testing of Real-World Embedded Systems Software. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, 309–326. https://www.usenix.org/conference/usenixsecurity18/presentation/corteggiani

[8] George Fairbanks. 2010. *Just enough software architecture: a risk-driven approach.* Marshall & Brainerd.

[9] Maayan Goldstein and Itai Segall. 2015. Automatic and Continuous Software Architecture Validation. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, Vol. 2. 59–68. https://doi.org/10.1109/ICSE.2015.135 ISSN: 1558-1225.

[10] Nikhil Krishna Gopalakrishna, Dharun Anandayuvaraj, Annan Detti, Forrest Lee Bland, Sazzadur Rahaman, and James C Davis. 2022. "If security is required": Engineering and Security Practices for Machine Learning-based IoT Devices. In *2022 IEEE/ACM 4th International Workshop on Software Engineering Research and Practices for the IoT (SERP4IoT)*. IEEE, 1–8.

[11] J.H. Jahnke, J. Niere, and J. Wadsack. 2000. Automated quality analysis of component software for embedded systems. In *8th International Workshop on Program Comprehension Proceedings IWPC 2000*. 18–26. https://doi.org/10.1109/WPC.2000.852476 ISSN: 1092-8138.

[12] Harika Kotha and V. Gupta. 2018. IoT Application, A Survey. *International Journal of Engineering & Technology* 7 (March 2018), 891. https://doi.org/10.14419/ijet.v7i2.7.11089

[13] Renata Martins Gomes and Marcel Baunach. 2021. A Study on the Portability of IoT Operating Systems. In *2021 GI Fachgruppentreffen Betriebssysteme (FGBS)*. Gesellschaft für Informatik e.V., Virtuell, Germany. https://doi.org/10.18420/fgbs2021f-01 Accepted: 2021-03-11T10:45:36Z.

[14] Marius Muench, Jan Stijohann, Frank Kargl, Aurelien Francillon, and Davide Balzarotti. 2018. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In *Proceedings 2018 Network and Distributed System Security Symposium (NDSS)*. Internet Society, San Diego, CA. https://doi.org/10.14722/ndss.2018.23166

[15] Phoenix Systems. 2021. Phoenix-RTOS. https://phoenix-rtos.com/

[16] Santonu Sarkar, Girish Maskeri Rama, and Shubha R. 2006. A Method for Detecting and Measuring Architectural Layering Violations in Source Code. In *2006 13th Asia Pacific Software Engineering Conference (APSEC'06)*. 165–172. https://doi.org/10.1109/APSEC.2006.7 ISSN: 1530-1362.

[17] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. 2022. Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, 1239–1256. https://www.usenix.org/conference/usenixsecurity22/presentation/scharnowski

[18] Dietmar Schreiner, Gergö Barany, Markus Schordan, and Jens Knoop. 2013. Comparison of type-based and alias-based component recognition for embedded systems software. *International Journal on Software Tools for Technology Transfer* 15, 1 (Feb. 2013), 41–52. https://doi.org/10.1007/s10009-012-0251-0

[19] Mingjie Shen. 2023. Null Pointer Dereference. https://github.com/zephyrproject-rtos/zephyr/pull/54847.

[20] Mingjie Shen, James C. Davis, and Aravind Machiry. 2023. *Towards Automated Identification of Layering Violations in Embedded Applications (WIP)*. https://doi.org/10.5281/zenodo.7921796

[21] Chad Spensky, Aravind Machiry, Marcel Busch, Kevin Leach, Rick Housley, Christopher Kruegel, and Giovanni Vigna. 2020. TRUST.IO: Protecting Physical Interfaces on Cyber-physical Systems. In *2020 IEEE Conference on Communications and Network Security (CNS)*. 1–9. https://doi.org/10.1109/CNS48642.2020.9162246

[22] Chad Spensky, Aravind Machiry, Nilo Redini, Colin Unger, Graham Foster, Evan Blasband, Hamed Okhravi, Christopher Kruegel, and Giovanni Vigna. 2021. Conware: Automated Modeling of Hardware Peripherals. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security (ASIA CCS '21)*. Association for Computing Machinery, New York, NY, USA, 95–109. https://doi.org/10.1145/3433210.3437532

[23] Jayashree Srinivasan, Sai R Tanksalkar, Paschal C Amusuo, James C Davis, and Aravind Machiry. 2023. Towards Rehosting Embedded Applications as Linux Applications. In *53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

[24] Neil R Storey. 1996. *Safety critical computer systems.* Addison-Wesley Longman Publishing Co., Inc.

[25] The Apache Software Foundation. 2022. Apache NuttX. https://nuttx.apache.org/

[26] The Linux Foundation. 2023. Zephyr® Project. https://www.zephyrproject.org/

[27] Travis Vachon. 2019. Whole Program LLVM. https://github.com/travitch/whole-program-llvm.

[28] Elecia White. 2011. *Making Embedded Systems: Design Patterns for Great Software.* "O'Reilly Media, Inc.". Google-Books-ID: No-BrEAAAQBAJ.

[29] Sunny Wong, Yuanfang Cai, Miryung Kim, and Michael Dalton. 2011. Detecting software modularity violations. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*. ACM, Waikiki, Honolulu HI USA, 411–420. https://doi.org/10.1145/1985793.1985850

[30] Vladimir Yussupov, Uwe Breitenbücher, Ayhan Kaplan, and Frank Leymann. 2020. SEAPORT: Assessing the Portability of Serverless Applications:. In *Proceedings of the 10th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and Technology Publications, Prague, Czech Republic, 456–467. https://doi.org/10.5220/0009574104560467

[31] Zephyr® Project, a Linux Foundation Project. 2021. Application Portability made easy with Zephyr OS and NXP (Webinar). https://www.zephyrproject.org/event/application-portability-made-easy-with-zephyr-os-and-nxp-webinar/

Received 2023-03-16; accepted 2023-04-21