POLITECNICO DI TORINO
Repository ISTITUZIONALE

Real-Time Control-Flow Integrity for Multicore Mixed-Criticality IoT Systems

Original
Real-Time Control-Flow Integrity for Multicore Mixed-Criticality IoT Systems / Eftekhari Moghadam, Vahid; Prinetto, Paolo; Roascio, Gianluca. - ELETTRONICO. - (2022), pp. 1-4. (Intervento presentato al convegno 2022 IEEE European Test Symposium (ETS) tenutosi a Barcelona (ESP) nel 23-27 May 2022) [10.1109/ETS54262.2022.9810441].

Availability:
This version is available at: 11583/2969412 since: 2022-07-04T14:07:22Z

Publisher:
Institute of Electrical and Electronics Engineers Inc.

Published
DOI:10.1109/ETS54262.2022.9810441

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2022 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

24 April 2024

Real-Time Control-Flow Integrity for Multicore Mixed-Criticality IoT Systems

Vahid EFTEKHARI MOGHADAM
Politecnico di Torino
CINI Cybersecurity National Lab.
Turin, Italy
vahid.eftekhari@polito.it

Paolo PRINETTO
Politecnico di Torino
CINI Cybersecurity National Lab.
Turin, Italy
paolo.prinetto@polito.it

Gianluca ROASCIO
Politecnico di Torino
CINI Cybersecurity National Lab.
Turin, Italy
gianluca.roascio@polito.it

***Abstract*—The spread of the Internet of Things (IoT) and the use of smart control systems in many mission-critical or safety-critical applications domains, like automotive or aeronautical, make devices attractive targets for attackers. Nowadays, several of these are mixed-criticality systems, i.e., they run both high-criticality tasks (e.g., a car control system) and low-criticality ones (e.g., infotainment). High-criticality routines often employ Real-Time Operating Systems (RTOS) to enforce hard real-time requirements, while the tasks with lower constraints can be delegated to more generic-purpose operating systems (GPOS).**

Much of the control code for these devices is written in memory-unsafe languages such as C and C++. This makes them susceptible to powerful binary attacks, such as the famous Return-Oriented Programming (ROP). Control-Flow Integrity (CFI) is the most investigated security technique to protect against such threats. At now, CFI solutions for real-time embedded systems are not as mature as the ones for general-purpose systems, and even more, there is a lack of in-depth studies on how different operating systems with different security requirements and timing constraints can coexist on a single multicore platform.

This paper aims at drawing attention to the subject, discussing the current scientific proposal, and in turn proposing a solution for an optimized asymmetric verification system for execution integrity. By using an embedded hypervisor, predefined cores could be dedicated to only high or low-criticality tasks, with the high-priority core being monitored by the lower-criticality core, relying on offline binary instrumentation and a light exchange of information and signals at runtime. The work also presents preliminary results about a possible implementation for multicore ARM platforms, running both RTOS and GPOS, both in terms of security and performance penalties.

***Index Terms*—IoT, internet of things, security, software security, control-flow integrity, operating systems, return-oriented programming, ROP, JOP**

I. Introduction

In recent times, the process of digitizing goods and services used by citizens appears to be inexorable. The number of connected embedded devices through the so-called Internet of Things (IoT) has a progression that projects it to nearly triple during the decade 2020-2030, reaching 25 billions [1]. By that date, 8 billions of consumer devices for internet/multimedia, such as smartphones, are expected to be running, and other use cases that will account for one billion devices will be smart logistics, smart infrastructures, and smart vehicles.

Such next-generation smart control systems are characterized by complex computing workloads, composed of tasks that are also very different from each other. On smart cars, for example, the onboard system consists of at least two fundamental parts. On the one hand, it must offer a series of graphic functions to the user through its touch screen, interact with the video cameras and the multimedia system, and others. Therefore, it needs advanced libraries and services typically integrated into a general-purpose operating system (GPOS). On the other hand, it must monitor and process in real time the data coming from the sensors equipping the physical parts of the machine, activate the ABS, or perform other critical routines with precise periods of execution: for this, it also needs to employ real-time operating systems (RTOS) for the correct scheduling of tasks. What is outlined is therefore a mixed-criticality (MC) software system, which requires computing platforms able to support this heterogeneity. In many cases, this is solved by adopting a hypervisor capable of creating divided compartments over a single platform, with operating systems dedicated to supporting tasks with different requirements [2] [3].

In such a diffused digitalization scenario, the attack exposition surface becomes extremely vast and appealing to malicious intruders, possibly interested in stealing sensitive data, or in controlling/interrupting the provision of many essential services. Beyond communications, applications and operating systems themselves are at risk, as very often the control code for these devices is written using the C language, which is still the second most used language for embedded programming in 2021 [4]. Although C offers high optimization capabilities, the free use of memory pointers can lead to the introduction of severe data corruption vulnerabilities [5] [6]. Because of these, powerful binary attacks can be carried out, such as Return-Oriented Programming (ROP) [7], which can force victim devices to execute completely arbitrary malware [8].

Control-Flow Integrity [9] has been presented in literature as the most promising technique to counter such a kind of threats. The basic idea is to monitor at runtime each control-flow transfer inside an application, by checking it against a predefined model of allowed branches, the Control-Flow Graph (CFG), obtained through offline static analysis of the source or binary code. The literature has been lavish with solutions of different nature for general-purpose systems [10] [11] [12], and even on a commercial level, Intel has introduced its own Control-Flow Enforcement Technology in its chips

[13]. On the other hand, the state of the art for special-purpose systems is different. Despite the proposals, a stable framework is still lacking, and additional problems on CFI feasibility are to be considered. The introduction of Pointer Authentication starting from ISA version 8.3 by ARM [14] is certainly a positive step, but still not complete and, in any case, not enabling for the many legacy applications based on microcontrollers in consolidated use.

Issues are related, for example, to the presence of preemptive schedulers that may interrupt the tasks of an RTOS at any instant, thus invalidating the results of the offline static analysis. Other issues derive from the limited resource availability or from the strict timing requirements, which collide with the request for additional times for integrity checks. As a strategy to overcome these limitations, one could think of exploiting the possibilities of a mixed-criticality system, with an operating system with laxer constraints that can allocate tasks for monitoring the real-time operating system, and with a hypervisor that provides the necessary protection guarantees for the additional memory parts needed to execute the integrity checks.

This paper intends to illustrate this solution concept, which is currently under study for multicore ARM platforms with hypervisor and general-purpose and real-time domains. Before entering into details, the paper offers a proper background on the topic. At the end, conclusions are drawn, and the work to be done to reach a quantified evaluation of the proposal is detailed.

II. Threat Model

For the purposes of our research, it is assumed the presence of a powerful attacker that can exploit several vulnerabilities in the code of an embedded device. Thanks to them, he/she is able to read and write its data memory. The code is assumed to be stored in Flash and not writable at runtime, while the data memory is assumed to be not executable as protected by basic Write-XOR-Execute policy [15]. Therefore, the attacker is for example capable of finding inside the executable code a series of segments referred to as *gadgets*, made up of very few instructions and all ending with a return or indirect jump machine instruction [7]. Then, he/she can inject into the memory a list of gadget addresses, and once the function returns, or a first *dispatcher branch* is taken [16], the gadget chain execution activates, and one after the other, all gadgets are executed on the victim device.

III. Control-Flow Integrity Principles

The concept behind Control-Flow Integrity [9] is monitoring the program at runtime to detect abnormal diversion from what is stated in its Control- Flow Graph (CFG). This is a directed graph where vertices are linear sequences of instructions having no branches except at their end (namely *basic blocks*), and edges are control-flow transfers that connect such blocks. The CFG is computed ahead of the execution, through static analysis of the source or the binary code, or through execution profiling. This is commonly referred to as the *offline phase*. The *online phase* represents instead the runtime protection, aimed at verifying that branches individuated by the offline phase are actually respected during the execution. To enforce this, an entity must be instructed with the CFG information at offline, and must be able to detect any violation at online (namely, the *CFI monitor*). *Binary instrumentation* techniques imply that the verification is performed by additional instructions, inserted in the online phase, and then performed by the application itself (or by an external monitor) at runtime, which rely on a system of labels that uniquely identify the destination of a branch. The operations that are going to be instrumented are the *indirect* control-flow transfers, i.e., branches with a non-constant argument, as they are the only ones that can be tampered with (namely, branch or call instructions with a register operands, return instructions, or any other instruction altering the value of the program counter).

One of the main drawbacks of using a complete CFG is the possible overhead. For that reason, some solutions implement a simplified CFG, grouping basic blocks with similar characteristics in a single vertex (e.g., entire functions). This is *coarse-grained* CFI. *Fine-grained* CFI policies are instead aimed at providing a fully-precise CFG, complete with every possible valid target of every indirect branch.

A large number of CFI solutions exist for general-purpose systems [10] [11] [17] [12] [18] [13]. Unfortunately, the same level of maturity is not found for real-time systems. The amount of additional resources required for protection is in most cases unaffordable for these devices, and a deep tradeoff on verification completeness towards coarse-grained approaches leads to high risks from a security point of view. Furthermore, if a degradation of execution times for CFI checks can be tolerated in the GPOS domain, this does not apply for RTOS, where computation tasks need to be completed within precise times.

For these reasons, an ideal solution to guarantee CFI for RTOS applications is comprised of the following features:

- **Full forward and backward branches protection** (a.k.a. *fine-grained* CFI);
- **Protection of the interrupt context**, since hardware interrupts can occur at any time, causing the execution of a handling routine that cannot be predicted by any static analysis [19];
- **Uncompromised workload schedulability**, i.e., ideally negligible overhead, considered both in terms of performance and code size.

IV. Asymmetric Multicore Control-Flow Verification

Due to the growing need to diversify the tasks of integrated systems, and to improve performance by avoiding frequency scaling, the most reasonable choice for new embedded applications seems to be adopting multicore architectures [20], possibly with hypervisors that support the creation of statically-divided application domains [2]. As already introduced, it is desirable to drive the community to investigate solutions exploiting the presence of multiple cores to guarantee CFI to the part of the system that performs highly-critical tasks by taking advantage of parallelization, without charging the cost of the checks to the RTOS.

A hypervisor (such as Bao [3]) can be configured to statically assign physical resources to two separate virtual machines, one to host the RTOS and one to host the GPOS, each running on a dedicated CPU and having its portion of memory. In addition, the hypervisor can reserve a portion of its storage to act as shared memory between the two cores (Figure 1). The RTOS with its tasks can be considered as a single binary to be instrumented according to the techniques described above. The enforcement is carried out according to the methodology described in [21], through a system of unique coupled labels, referring to the source and destination of the branches. In addition to this, labels are also created to uniquely identify the caller of a procedure, to enforce backward edges. Finally, to support interrupt and scheduler awareness, a secure saving of context registers is applied to the input of interrupt handling routines. The algorithm used for the instrumentation is the one described in [22].

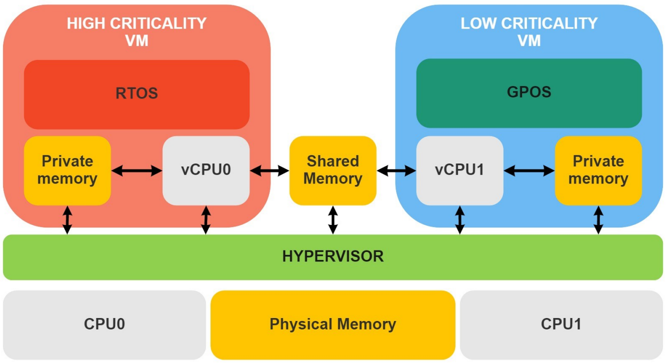

Fig. 1: The described architecture for supporting multicore control-flow verification.

Instrumentations do not contain validation code, but are represented by a minimal group of instructions that are only needed to transfer sensitive information (concerning the position in the code or the value of a context register) on the shared memory. At that point, through an atomic signal to the hypervisor, the monitoring task in the context of the GPOS on the other core is invoked and executed.

The monitor task in the GPOS partition has the priority to preempt all other tasks when invoked. Therefore, the checks are carried out in parallel with the critical execution on the other side, without compromising its schedulability. However, these occur in a few clock cycles after the branch, thus managing to stop any attack attempt in the bud. The checks are based on 3 main memory sections present in the private storage of the task monitor: (i) a table containing the allowed edges, in the form of coupled source-destination labels, accessed as a ROM hash table [21], (ii) a label call stack to store destination labels associated with calls and verify the identity of the last caller on return, and (iii) a register call stack to hold context in the event of a sudden interrupt.

When allocating memories of the two OS, the hypervisor takes care of duly protecting the shared memory portion (to have unidirectional flow and not to be used outside the protocol) and the private memory portion of the monitor task (not to be accessed by other tasks).

At now, this technique has only been evaluated in theory, based on rough estimates of the execution times of the ARM instructions necessary for the instrumentation. To these, the time of the *Virtual Inter Processor Interrupt* (vIPI) managed by the hypervisor must be added, which can have variable times depending on the implementation, the platform, and its support for virtualization in hardware. Future work and tests need to be done in this regard.

V. Related Work

As already introduced, the current scientific proposal of CFI solutions for RTOS, in addition to not being remarkably rich, presents a series of problems mostly related to scheduling and the support capabilities of resource-constrained platforms. All the examinated solutions incurred in the trade-off between security and overhead, although in different ways and to different degrees.

In RECFISH [23], integrity checks are implemented through binary instrumentation for forward branches and the use of a shadow stack [24] for backward branches. Edges are replaced by trampolines towards the instrumentation code, which is located in a special section. Although it has high coverage as it is fine-grained, RECFISH appears to have no protection for interrupts, which run in privileged mode and are thus a possible vulnerability [19]. Plus, experiments showed that 15% of workloads were not schedulable anymore after the instrumentation, with an overhead of 30% for the worst cases.

In TrackOS [25], a monitoring task is created and scheduled along with all the other "normal" tasks. Therefore, CFI checks only happen when scheduled, depending on the workload definition, instead of happening at every vulnerable control-flow transfer. While it does bring security enhancements with minimal impact on workload schedulability, TrackOS does not provide a sufficient level of security: no control-flow violation is detected as long as no context switch to the monitoring task occurs.

Then, solutions that make use of the Pointer Authentication (PA) facility introduced by ARM [14] have been proposed. This new feature adds a hardware-assisted way to sign pointers with an unreadable unique code, that will be used for authentication before consuming them. Camouflage [26] uses PA to sign function pointers and return addresses to protect, respectively, forward and backward edges in kernel code. Other than a non-negligible overhead ranging from 10% to 30%, Camouflage also lacks interrupt context protection. Authentication and branching are not done as an atomic operation, but are instead executed by separate instructions, creating the possibility of time-of-check-to-time-of-use (TOCTOU) attacks [27].

PATTER (Pointer AuThenTication for kERnel) [28] works together with LLVM to insert instrumentation right at its

Intermediate Representation level and protect both forward and backward branches. All pointers used for forward branches are signed and then stored in memory, and when they need to be used, they are loaded from memory and then authenticated. When branching, the `blraa` instruction is used to guarantee atomicity. That instruction authenticates the pointer before a *branch-and-link* operation, making TOCTOU attacks impossible to be executed. As for backward edges, prologues to sign the return value are put at the start of each function, and to return the atomic `retaa` is used, to authenticate and branch in a single time. Also for PATTER, the overhead reaches 20%. Other problems arise from the the possible presence of several corner-cases regarding function pointer generation, like the use of pointer arithmetic, pointers holding physical addresses, and pointers inside of unions. All of these cases are not correctly handled by the algorithm, so they need specific workarounds which need to be kernel-dependent.

VI. Conclusions

This paper has presented a research hot topic in the IoT domain, on the protection of real-time software against binary attacks in a multicore environment where a general-purpose partition also exists. The paper has offered bullets on the context and on the importance of an investigation in this sense. After that, preliminary arguments made on a possible application of the solution concept have been done. To give greater vigor to the study, the results of the practical implementation of the technique are expected, from which it will be possible to deduce the real overhead cost on the system, as well as the actual security enhancement based on the results of standard evaluation benchmark execution [29].

VII. Acknowledgments

The work described in this paper is partially supported by (i) the European Union's Horizon 2020 research and innovation programme, under grant agreement No. 830892, project SPARTA, (ii) the Ph.D. research program of TIM S.p.A (Italy).

References

[1] A. Holst, "Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2030." https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/, 2021. [Online; Accessed December 14, 2021].

[2] Accelerat, "The CLARE Software Stack." https://accelerat.eu/clare, 2021. [Online; Accessed December 14, 2021].

[3] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto, "Bao: A lightweight static partitioning hypervisor for modern multi-core embedded systems," in *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[4] I. Spectrum, "Top Programming Languages." https://spectrum.ieee.org/top-programming-languages/, 2021. [Online; December 14, 2021].

[5] "CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer." https://cwe.mitre.org/data/definitions/119.html, 2021. [Online; December 14, 2021].

[6] "CWE-825: Expired Pointer Dereference." https://cwe.mitre.org/data/definitions/825.html, 2021. [Online; December 14, 2021].

[7] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, p. 2, 2012.

[8] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the expressiveness of return-into-libc attacks," in *International Workshop on Recent Advances in Intrusion Detection*, pp. 121–141, Springer, 2011.

[9] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.

[10] T. Bletsch, X. Jiang, and V. Freeh, "Mitigating code-reuse attacks with control-flow locking," in *Proceedings of the 27th Annual Computer Security Applications Conference*, pp. 353–362, ACM, 2011.

[11] Yubin Xia, Yutao Liu, H. Chen, and B. Zang, "Cfimon: Detecting violation of control flow integrity using performance counters," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pp. 1–12, June 2012.

[12] M. Zhang and R. Sekar, "Control flow integrity for cots binaries," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pp. 337–352, 2013.

[13] V. Shanbhogue, D. Gupta, and R. Sahita, "Security analysis of processor instruction set architecture for enforcing control-flow integrity," in *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, pp. 1–11, 2019.

[14] Q. Technologies, "Pointer authentication on armv8.3." https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf, 2017.

[15] P. Team, "PaX Non-Executable Pages Design and Implementation." https://pax.grsecurity.net/docs/noexec.txt, 2003. [Online; December 15, 2021].

[16] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pp. 30–40, ACM, 2011.

[17] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Securing untrusted code via compiler-agnostic binary rewriting," in *Proceedings of the 28th Annual Computer Security Applications Conference*, pp. 299–308, 2012.

[18] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque control-flow integrity.," in *NDSS*, vol. 26, pp. 27–30, 2015.

[19] N. Maunero, P. Prinetto, and G. Roascio, "Cfi: Control flow integrity or control flow interruption?," in *2019 IEEE East-West Design Test Symposium (EWDTS)*, pp. 1–6, Sep. 2019.

[20] G. Kornaros, *Multi-core embedded systems*. CRC Press, 2018.

[21] N. Maunero, P. Prinetto, G. Roascio, and A. Varriale, "A fpga-based control-flow integrity solution for securing bare-metal embedded systems," in *2020 15th Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, pp. 1–10, 2020.

[22] V. Forte, N. Maunero, P. Prinetto, and G. Roascio, "Prolepsis: Binary analysis and instrumentation of iot software for control-flow integrity," in *2021 International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME)*, pp. 1–6, 2021.

[23] R. J. Walls, N. F. Brown, T. Le Baron, C. A. Shue, H. Okhravi, and B. C. Ward, "Control-flow integrity for real-time embedded systems," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[24] A. Francillon, D. Perito, and C. Castelluccia, "Defending embedded systems against control flow attacks," in *Proceedings of the first ACM workshop on Secure execution of untrusted code*, pp. 19–26, ACM, 2009.

[25] L. Pike, P. Hickey, T. Elliott, E. Mertens, and A. Tomb, "Trackos: A security-aware real-time operating system," in *International Conference on Runtime Verification*, pp. 302–317, Springer, 2016.

[26] R. Denis-Courmont, H. Liljestrand, C. Chinea, and J.-E. Ekberg, "Camouflage: Hardware-assisted cfi for the arm linux kernel," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2020.

[27] R. P. Abbott, J. S. Chin, J. E. Donnelley, W. L. Konigsford, S. Tokubo, and D. A. Webb, "Security analysis and enhancements of computer operating systems," tech. rep., National Bureau of Standards, Washington DC Institute for Computer Sciences, 1976.

[28] Y. Yang, S. Zhu, W. Shen, Y. Zhou, J. Sun, and K. Ren, "Arm pointer authentication based forward-edge and backward-edge control flow integrity for kernels," *arXiv preprint arXiv:1912.10666*, 2019.

[29] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "Ripe: Runtime intrusion prevention evaluator," in *Proceedings of the 27th Annual Computer Security Applications Conference*, pp. 41–50, 2011.