Control-Flow Integrity for Real-Time Operating Systems: Open Issues and Challenges

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

Marco MELONI
Politecnico di Torino
Turin, Italy
marco.meloni@studenti.polito.it

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

***Abstract*—The pervasive presence of smart objects in almost every corner of our everyday life urges the security of such embedded systems to be the point of attention. Memory vulnerabilities in the embedded program code, such as buffer overflow, are the entry point for powerful attack paradigms such as Code-Reuse Attacks (CRAs), in which attackers corrupt systems' execution flow and maliciously alter their behavior. Control-Flow Integrity (CFI) has been proven to be the most promising approach against such kinds of attacks, and in the literature, a wide range of flow monitors are proposed, both hardware-based and software-based. While the formers are hardly applicable as they impose design alteration of underlying hardware modules, on the contrary, software solutions are more flexible and also portable to the existing devices. Real-Time Operating Systems (RTOS) and their key role in application development for embedded systems is the main concern regarding the application of the CFI solutions.**

This paper discusses the still open challenges and issues regarding the implementation of control-flow integrity policies on operating systems for embedded systems, analyzing the solutions proposed so far in the literature, highlighting possible limits in terms of performance, applicability, and protection coverage, and proposing possible improvement directions.

***Index Terms*—security, cybersecurity, code-reuse attacks, ROP, control-flow integrity, embedded systems, operating systems**

I. Introduction

The widespread use of the Internet of Things (IoT) and the social impact of the IoT applications and services, obliges data protection and information security. Due to its heterogeneous nature and complex architecture, IoT infrastructure and the devices operating within, from large servers in the cloud to small edge computing devices, are always facing security and privacy issues. Thus, effective and powerful protection mechanisms are necessary to be implemented in order to guarantee the reliability and safety of operations. Security of IoT can be classified under two main categories: security in isolation and security inside the cloud. Security measures inside the cloud focus more on the IoT infrastructure and the security of its services. Issues like device authentication, secure network access, secure communication, storage, and availability fall into this category, whereas security in isolation targets the solutions designed around the operational safety of devices. Issues like secure boot, secure update, and memory isolation of processes are examples of this category.

Trustworthiness is a key part of achieving higher security for embedded systems. Aspect such as adopted programming language can play an important role. Due to their nature, most of the code written for the embedded devices is in C and C++ [3]. These languages allow better low-level control over the hardware and its resources, such as direct memory management. Operating with such a level of freedom could cause a wide range of security vulnerabilities. For example, out-of-bound memory writes (e.g., *buffer overflow*) [4] can lead to data corruption on stack or heap, and an adversary can exploit these vulnerabilities in absence of countermeasures to inject malicious content into the memory. Some of these vulnerabilities also enable attackers to corrupt *code pointers*, which are used as an argument to indirect control-flow instructions. In such exploitations, the attackers redirect the program execution to a portion of code that is already present in the memory (e.g., part of the standard library) instead of directly injecting malware. Randomly in memory, there are short sequences of instructions called *gadgets* that an adversary can use. By chaining multiple gadgets together, one can force a complete arbitrary execution. This kind of threat is usually referred to as *Code-Reuse Attacks* (CRA), and well-known exploit paradigms such as Return Oriented Programming (ROP) [50] [20] [23] [48] and Jump Oriented Programming (JOP) [16] [26] are based on such techniques.

Control-Flow Integrity (CFI) is an ideal technique that can be enforced to prevent attacks that alter the program execution flow [8]. CFI enforcement dictates that the program execution should follow a predetermined path following the Control-Flow Graph (CFG). The CFG can be obtained by analyzing the program binary. Several solutions for the CFI have been proposed ranging from hardware-based solutions to pure software implementations. For example, some commercial tools such as the LLVM [5] compiler partially implement security policies introduced by CFI in software. Although the hardware-based solutions are the finest, they lack applicability in most cases. On the contrary, software-based solutions are more portable. Embedded systems nowadays have RTOSs or embedded OSs on board since they help in producing better code by providing the common functionalities needed for application development and reducing the time and effort of the development by hiding away underlying hardware details.

Although the adoption of CFI solutions is increasing among commercial software, the embedded world has not benefited much. Due to resource constraints of embedded platforms, CFI solutions are rare, and performance or applicability costs prevent easy integration.

This paper discusses on challenges that are to be faced when proposing an all-encompassing solution, analyzing issues of the current proposed solution for CFI for embedded operating systems. The remainder of the paper is organized as follows: Section II provides some technical background on Control-Flow Integrity (CFI); Section III analyses in detail CFI solutions proposed for embedded systems, their issues, and challenges to be addressed. Section IV finally concludes the paper.

II. Background

Arbitrary Code Execution (ACE) in computer security is the term used to express the attacker's ability to execute arbitrary programs or code on a target machine. There are different classes of vulnerability [6] [1] [2] that can be used by an adversary to exploit security flaws and accomplish ACE. Memory safety vulnerabilities such as buffer overflow [47] are among the most common ones in the embedded world due to the heavy use of system programming languages like C/C++. Although these languages provide better means of low-level control over hardware resources, especially memory, improper management by programmers can introduce security vulnerabilities [47] [11] [53].

Arbitrary code execution exploits the control over the value of the *instruction pointer* (also known as *Program Counter*) of the running program. The instruction pointer points to the next instruction that will be executed, therefore by tampering with the value of IP, attackers can alter the execution and redirect it to malicious code, referred to as *payload*. Classically, vulnerabilities present in stack [45] allowed the attackers to inject the malicious code including the corrupted instruction pointer in the program's memory [45]. Due to the adoption of protection mechanisms such as *stack canary* [29] or *Data Execution Prevention* (DEP) [52] code injection exploits lost their relevance. For example, in stack canary protection mechanism when a function call takes place, some know values (e.g., a random value) are pushed on top of the stack placed between the return address and the function's local variables. On the return from the function, these values are checked and if they have been changed, it is considered as an attack resulting in program termination. With these protection mechanisms in place, a new paradigm of attacks emerged, the so-called *Code-Reuse Attacks* (CRA). In CRA, attackers exploit the program's code present in the memory, bypassing the protection mechanism (e.g., DEP) and altering the programs' execution flow for getting a malicious result. For example, *Return-Oriented Programming* (ROP) [20] [34] [24] [23] [40] is a class of CRA in which short code sequences within the existing binary, referred to as *gadgets*, ending with `ret` instructions, are linked together and executed in arbitrary order by exploiting the stack (assuming the return address can point anywhere) to hijack the control flow. By having a relatively large enough codebase, it has been shown that one can link enough gadgets to achieve a meaningful result with Turing-compute capabilities [50] [54]. Other classes of CRA attacks exploit other forms of gadgets like indirect `jmp` and `call` to achieve the desired control-flow alteration. *Jump-Oriented Programming* (JOP) [16] [26] and *Call-Oriented Programming* (COP) [49] are examples of that. To tackle the CRA exploits, many studies and research have been done and some countermeasures such as *Address Space Layout Randomization* (ASLR) [14], *Shadow Call Stack* [55] [35] [27] [19] [18], or heuristic-based approaches like DROP [25].

In the paper [7], the concept of *Control-Flow Integrity* (CFI) as a general defense technique for control-flow hijacking attacks is introduced. CFI security policy ensures that the program execution flow strictly follows the path of *Control-Flow Graph* (CFG). The CFG in question can be defined statically through analysis (source-code analysis, binary analysis, or execution profiling) before program execution [8]. CFG represents the control flow of a program by grouping non-jumping instructions which are executed sequentially into basic blocks (the graph *nodes*), and the branches caused by `jump`, `call`, or `ret` instructions (the graph *edges*). Subsequently, at runtime, program control-flow transfers (potentially corrupted due to an attack) are checked against the CFG to ensure correct execution flow. Since in most cases it is possible to assume the code stored in Flash memory as immutable, monitoring is only enforced on indirect forms of branching instructions.

Given a CFG, CFI policy enforcements are generally categorized into *coarse-grained* or *fine-grained* types. *Coarse-grained* polices are those who do not strictly follow the CFG due to lack of precision, and they are more like heuristic solutions chasing against most probable cases. The possible aim of such solutions could be tackling the performance overhead. Contrarily, *fine-grained* solutions comply strictly with the CFG. Thus, fine-grained CFI policies are the only solutions that can guarantee full compliance with the intended program design.

In literature, there are many CFI implementations purposed each focusing on different aspects such as precision, performance, or scalability. There exist software solutions based on binary instrumentation such as *label-based binary instrumentation*, first proposed by the original CFI paper [7], or *Control-Flow Locking* [15]. Also, hardware-based solutions tackle the monitoring through hardware with the hope of mitigating the overhead, such as the solution introduced by Sullivan *et al.* [51] on SPARC LEON 3 processor. In the next section, an analysis will be made of some of the relevant techniques proposed to provide real-time operating systems with CFI protection, and their effective applicability and possible limitations will be discussed, including some suggestions for future studies that aim to find an all-encompassing and improved solution compared to the techniques proposed so far.

III. Current Solutions and Open Issues

To the authors' best knowledge, current research on the implementation of CFI for embedded and real-time operating systems is poor, especially regarding software-based solutions for commercially available systems. However, some research for commodity operating systems can be taken into consideration as a starting point for application to RTOSs.

An ideal, all-encompassing solution for real-time applications should have these characteristics:

- **Complete CFI coverage**, which is comprised of:
  - **Full forward and backward branches protection**;
  - **Protection of the interrupt context**, which is not predictable from the CFG analysis only [44].
- **Uncompromised workload schedulability**: this means having ideally negligible overhead. This is evaluated depending on the specific system and workload. Most embedded platforms running real-time applications are usually resource-constrained, which makes this a significant issue. Overhead should be considered both for performance and code size.

This ideal solution is difficult to achieve because it incurs the trade-off between security and overhead. All proposed solutions at the moment have to compromise on one of them, although to different degrees, while giving priority to the other one. This makes RTOS hardening a still open issue.

Among the solutions providing full forward and backward branch protection, one possibility is RECFISH [56]. In this RTOS-specific solution, a traditional approach to CFI is applied, using static analysis to generate the complete CFG and then instrumenting the binary with label checking. The hardware memory protection functionalities of the ARM platform are employed to isolate a memory region for the shadow stack, which is used to protect backward branches. The system interacts with the shadow stack through supervisor calls that, by triggering a software interrupt, can write to the protected memory in privileged mode. While these mechanisms work at the bare-metal level, OS support is added by making some changes to FreeRTOS [36], a popular open-source RTOS, by modifying the scheduler, task initialization, and task control block. As expected from classical CFI solutions, there is significant overhead from all the extra instructions introduced by the binary instrumentation, with relevant schedulability difficulties. The researchers tested a wide array of workloads and the results were mixed depending on their composition: around 15% of all workloads tested were not schedulable anymore after instrumentation, and a 30% of loss in utilization was measured for the worst cases. As a possible way to mitigate this problem, the researchers propose the idea of marking the tasks that do not allow any user interaction as "safe" and not instrument those, while applying CFI checks only to the "unsafe" ones. In their benchmarks, this has been proven successful, but introduces the new problem of having the developer decide which tasks are safe or not and trust their judgement. Inter-task communication between "safe" and "unsafe" tasks also becomes a problem, along with state persistence among subsequent task executions. Creating a tool able to analyse the tasks and schedule and automatically mark tasks as "safe" or "unsafe" could be a way to significantly reduce the schedulability issue. Also, RECFISH is not completely secure, since it is still vulnerable to attacks to interrupt handlers [44], that operate at the same privilege level of supervisor calls and that can potentially modify the shadow stack memory.

A more secure solution that also involves protection against interrupt attacks can potentially be found looking at non-RTOS-specific research. Several solutions aiming to solve the problem of kernel security in commodity operating systems have been proposed, with approaches that can potentially be considered for embedded OSs as well. Some of them, like KCoFI [30], use coarse-grained CFI, which can be beaten with carefully-crafted attacks [37] [28] [22] [31]. Finding a fine-grained solution was thus made necessary to obtain adequate security guarantees. One proposed solution is FINE-CFI [41]: this approach uses virtualization as a way to protect not only the forward and backward branches through code instrumentation with indexed hooks [42], but the interrupt context as well: Linux KVM [39] handles a protected stack used to store control data whenever an interrupt is called, and check it when going back to the normal operations. This solution still introduces some non-negligible overhead, going from 10% to 20% depending on the benchmark. This is of course a source of issues for task schedulability in real-time situations. Despite the increasing support for KVM on ARM, especially from ARMv8, and even if KVM seems to introduce acceptable latencies for most real-time workloads [10] when compared to the past [12], there is still the problem of KVM being limited to use with Linux. Preempt RT [33] and LITMUS^{RT} [21], respectively a real-time patch and an extension to the stock kernel, can mitigate the latency problem present in the standard Linux kernel. In any case, the issue of the size and of a too-large Trusted Computing Base (TCB) still stands. This is a problem for memory-constrained systems, and translates into a very wide attack surface. An ideal solution would be able to support the adoption of a more lightweight RTOS, like FreeRTOS or RIOT, for use on resource-constrained systems. For those situations, having an embedded specific hypervisor that is not dependent on a single kernel architecture would be preferable.

Despite the advantages, both of the last two works presented cannot appropriately handle the schedulability problem. This issue is instead primarily addressed by TrackOS [46]. TrackOS is a RTOS based on FreeRTOS, that adds CFI monitoring capabilities while prioritizing determinism and workload schedulability. It works by generating a call graph for every regular task through static analysis, and then by scheduling a special "monitoring" task that gets scheduled along with the other ones as part of the workload. The task checks the control stack of each other tasks against their call graph for clues of control flow violations whenever the scheduler executes it. This allows the user to control the overhead introduced by CFI and only schedule the monitoring task in a way that does not compromise the schedulability of the overall workload, depending on deadlines and their relative importance compared to overall security protection. Even if providing security improvements in a very time-efficient manner, this approach is not a complete solution against control-flow attacks: there is no real control over branching operations, which means that, as long as a task is not switched out by the scheduler, any attack will go on undetected. More, a monitoring task just being a task like the others is also a problem, as it incurs the risk of being starved by a higher-priority task, if any exist. Overall, TrackOS is a perfect example of the trade-off between schedulability and full CFI coverage, achieving the former at the cost of a security loss.

An area of interest is the world of hardware-assisted support for security features, which is becoming more and more common in the embedded world as well. ARM in particular has been working in that direction for the last few years. With the introduction of ARMv8.3, *Pointer Authentication* (PA) [17] has been added as a hardware-assisted way of securing function pointers by signing them with a unique code that is to be used for authentication before consuming them. This code (called Pointer Authentication Code or PAC) is generated using the QARMA algorithm [13], which takes the pointer, a key, and a modifier to generate a value that will get truncated and inserted in the unused bits of the pointer. This is possible since 64-bit pointers are oversized for the address space they usually refer to, and thus have unused bits. There are 5 different keys that can be used, and the modifier depends on the developer's choice.

Since the first Qualcomm whitepaper came out in 2017 [38], several applications of PA have been proposed to implement control-flow protection through function pointer authentication instead of using labels. One of them is Camouflage [32], which protects the PA keys by using them through a function contained inside of a memory page that is mapped as a XOM (*eXecution Only Memory*) at the kernel level by the hypervisor, which also controls key use in such a way to avoid exposing them. The modifiers used for the PAC generation differ based on whether they are generated for forward or backward branches. The control-flow protection works by signing the function pointers for forward branches and the stack pointer to protect backward branches and then authenticating them before calling a function or returning from it. This is done through setter and getter inline functions in the first case and function prologues and epilogues in the second one. These work thanks to code instrumentation in a similar fashion to traditional labels, which means that this system is vulnerable to time-of-check-to-time-of-use attacks (TOCTOU) [9], and has no protection from interrupt-based attacks, which could be employed to expose and modify kernel registers, like modifiers or previously authenticated pointers. Additionally, while the risk of pointer-reuse attacks is mitigated through the use of custom modifiers, it is still an unsolved vulnerability. There is also the issue of not respecting ISO C compliance and needing to adapt some C library functions to be able for Camouflage to work. As for the overhead introduced by this system, system call benchmarks denote a performance overhead ranged from 10% to 30%, but the researchers argue that actual user applications are not that system-call intensive, and provide 3 benchmarks with an average of 7% overhead. The system call usage of tasks in a real-time schedule thus can influence heavily the total overhead.

A similar approach is used in PATTER (Pointer AuThenTication for kERnels) [57], which uses static analysis and code instrumentation to protect the kernel by leveraging the security functionalities of PA. The code analysis and instrumentation are applied to the Intermediate Result from the LLVM compilation process of Clang, and once again, custom modifiers are used to try to have unique context identifiers needed to make pointer-reuse attacks more difficult. Here, one can notice that all function pointers in the kernel memory are in the same address range: this means that the address of each one of them can be used as a unique identifier to make pointer-reuse attacks impossible. Compared to the previous work, this one mitigates the risk of TOCTOU attacks by using specific instructions like `brlaa` and `retaa`, which implement authentication and branching in a single instruction, guaranteeing the atomicity of the operation. This is unfortunately not possible for function-pointer store and load operations, but the authors argue that this is a non-issue, since they will be authenticated atomically before branching anyway. To get full coverage of the function pointers inside the kernel the static algorithm used by PATTER is not enough, since there are some special cases to consider, like the use of pointer arithmetic, pointers holding physical addresses, and pointers inside of unions. In the considered kernel (Linux), all of these are fairly rare. So, for the first two cases, manual patching is possible, and for the last one a protocol using the 64-bit alignment of pointers as a way to recognize the element type is a working solution. However, there is no guarantee this is the case for all OS kernels, and more robust/structured solutions to the problem would be ideal. Looking at the system call benchmarks, PATTER introduces around 10%-20% performing overhead. Additional testing would be needed to assess the actual impact on normal operation when running user applications.

Both PA-based solutions still result in introducing some overhead, although significantly less compared to other solutions, but only PATTER introduced adequate protection against TOCTOU attacks. The use of atomic instructions would be advisable for Camouflage as well. Also, the adoption of PA introduced new attack vectors, because of pointer substitution. The feasibility and dangerousness of those attacks still have to be adequately evaluated and would be a matter of interest to better frame PA as a security measure.

Table I summarizes the results of the analysis made on the solutions presented, with respect to the individuated metrics of interest: coverage of both backward and forward edges, interrupt awareness, and workload schedulability.

IV. Conclusions

The present paper analysed control-flow integrity solutions designed for embedded systems, focusing on embedded op-

TABLE I
SUMMARY OF ANALYSED SOLUTIONS AND THEIR COVERAGE OF THE MAIN ISSUES.

Solution	F&B coverage	Int. awareness	Schedulability
RECFISH [56]	Yes	No	Compromised
FINE-CFI [41]	Yes	Yes	Compromised
TrackOs [46]	No	No	Uncompromised with appropriate scheduling
Camouflage [32]	Yes	No	Compromised
PATTER [57]	Yes	Yes	Compromised

erating systems and real-time operating systems. As it was stated, the CFI solutions for the embedded world are limited. Due to resource/performance constraints, proposed solutions have many drawbacks that need to be addressed and their applicability is limited due to this fact. As an example, TrackOS can flexibly tackle the problem of scheduling, however, it lacks full security coverage. This trade-off between performance and security coverage is the main point of discussion. The ideal solution that should be considered has to guarantee the full coverage (forward edges, backward edges, and interrupts awareness) while introducing no significant overhead. How to achieve such a solution is still an open issue, especially considering the RTOS sector.

Some ideas and techniques from the current research could be taken into consideration for future solutions. Virtualization is a technique that has become more common in the embedded sector in recent years, with native support present in most recent ARM platforms. Hence, hypervisor-based techniques could be a viable solution. On that end, the limiting factor of current research is the common use of KVM. While fine for commodity operating systems, for real-time platforms it could be better to use some different virtualization techniques to be able to adopt lightweight RTOS, thus reducing both bloat and possibly latency. Tools such as Bao [43], an open-source lightweight hypervisor created for embedded platforms with specific consideration for real-time use, could be compelling instruments in that direction. Another interesting area is native hardware extensions, like PA. Having this hardware support for security features could be a very powerful ally when trying to reduce overhead while maintaining strong security guarantees. The inherent overhead from the authentication, even if limited, could still be a problem for real-time applications. In the case of RTOSs selecting tasks in a similar fashion to the optimizations proposed for RECFISH could be a way to reduce the overall delay and maintain better schedulability, by only applying PA to the tasks deemed “unsafe”.

V. ACKNOWLEDGMENTS

This paper is partially supported by the Ph.D. research program of TIM S.p.A (Italy).

REFERENCES

[1] Understanding type confusion vulnerabilities. https://www.microsoft.com/security/blog/2015/06/17/understanding-type-confusion-vulnerabilities-cve-2015-0336/, 2015. [Online; accessed 30-July-2021].

[2] Ghostscript RCE through type confusion. https://securitylab.github.com/research/cve-2018-19134-ghostscript-rce/, 2019. [Online; accessed 30-July-2021].

[3] Interactive: The Top Programming Languages 2020 - IEEE Spectrum. https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2020, 2020. [Online; accessed 07-June-2021].

[4] CCWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow'). https://cwe.mitre.org/data/definitions/120.html, 2021. [Online; accessed 21-July-2021].

[5] Clang 13 documentation - CONTROL FLOW INTEGRITY. https://clang.llvm.org/docs/ControlFlowIntegrity.html, 2021. [Online; accessed 27-July-2021].

[6] Deserialization of untrusted data. https://owasp.org/www-community/vulnerabilities/Deserialization_of_untrusted_data, 2021. [Online; accessed 30-July-2021].

[7] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[8] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.

[9] Robert P Abbott, Janet S Chin, James E Donnelley, William L Konigsford, S Tokubo, and Douglas A Webb. Security analysis and enhancements of computer operating systems. Technical report, NATIONAL BUREAU OF STANDARDS WASHINGTONDC INST FOR COMPUTER SCIENCES AND ..., 1976.

[10] Luca Abeni and Dario Faggioli. Using xen and kvm as real-time hypervisors. *Journal of Systems Architecture*, 106:101709, 2020.

[11] J. Afek and A. Sharabani. Dangling pointer: Smashing the pointer for fun and profit, 2007.

[12] Mehdi Aichouch, Jean-Christophe Prevotet, and Fabienne Nouvel. Evaluation of an rtos on top of a hosted virtual machine system. In *2013 Conference on Design and Architectures for Signal and Image Processing*, pages 290–297. IEEE, 2013.

[13] Roberto Avanzi. The qarma block cipher family. almost mds matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes. *IACR Transactions on Symmetric Cryptology*, pages 4–44, 2017.

[14] S. Bhatkar, D. DuVarney C, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *USENIX Security Symposium*, volume 12, pages 291–301, 2003.

[15] 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*, pages 353–362. ACM, 2011.

[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*, pages 30–40. ACM, 2011.

[17] ARM Connected Community Blog. Armv8-a architecture – 2016 additions. https://community.arm.com/groups/processors/blog/2016/10/27/armv8-a-architecture-2016-additions, 2016.

[18] C. Bresch, D. Hély, A. Papadimitriou, A. Michelet-Gignoux, L. Amato, and T. Meyer. Stack redundancy to thwart return oriented programming in embedded systems. *IEEE Embedded Systems Letters*, 10(3):87–90, Sep. 2018.

[19] C. Bresch, A. Michelet, L. Amato, T. Meyer, and D. Hely. A red team blue team approach towards a secure processor design with hardware shadow stack. In *2017 IEEE 2nd International Verification and Security Workshop (IVSW)*, pages 57–62, July 2017.

[20] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38. ACM, 2008.

[21] John M Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C Devi, and James H Anderson. Litmusˆ rt: A testbed for empirically comparing real-time multiprocessor schedulers. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 111–126. IEEE, 2006.

[22] N. Carlini and D. Wagner. Rop is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, 2014.

[23] S. Checkoway, L. Davi, A. Dmitrienko, A.R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 559–572. ACM, 2010.

[24] S. Checkoway, A. J. Feldman, B. Kantor, J.A. Halderman, E. W. Felten, and H. Shacham. Can dres provide long-lasting security? the case of return-oriented programming and the avc advantage. *EVT/WOTE*, 2009, 2009.

[25] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. Drop: Detecting return-oriented programming malicious code. In A. Prakash and I. Sen Gupta, editors, *Information Systems Security*, pages 163–177, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[26] P. Chen, X. Xing, B. Mao, L. Xie, X. Shen, and X. Yin. Automatic construction of jump-oriented programming shellcode (on the x86). In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 20–29. ACM, 2011.

[27] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis. Hcfi: Hardware-enforced control-flow integrity. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 38–49. ACM, 2016.

[28] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 952–963, 2015.

[29] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, , and H. Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. 98:5–5, 01 1998.

[30] John Criswell, Nathan Dautenhahn, and Vikram Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *2014 IEEE Symposium on Security and Privacy*, pages 292–307. IEEE, 2014.

[31] L. Davi, A. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416, 2014.

[32] Rémi Denis-Courmont, Hans Liljestrand, Carlos Chinea, and Jan-Erik Ekberg. Camouflage: Hardware-assisted cfi for the arm linux kernel. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[33] The Linux Foundation. PREEMPT_RT: The Linux Kernel real-time patch. https://wiki.linuxfoundation.org/realtime/start, 2021.

[34] A. Francillon and C. Castelluccia. Code injection attacks on harvard-architecture devices. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 15–26. ACM, 2008.

[35] Aurélien Francillon, Daniele Perito, and Claude Castelluccia. Defending embedded systems against control flow attacks. In *Proceedings of the first ACM workshop on Secure execution of untrusted code*, pages 19–26. ACM, 2009.

[36] FreeRTOS. FreeRTOS - Market Leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions. https://www.freertos.org/, 2021. [Online; accessed 27-July-2021].

[37] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *2014 IEEE Symposium on Security and Privacy*, pages 575–589, May 2014.

[38] Qualcomm Technologies Inc. Pointer authentication on armv8.3. Technical report, January 2017.

[39] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230. Dttawa, Dntorio, Canada, 2007.

[40] T. Kornau et al. *Return oriented programming for the ARM architecture*. PhD thesis, Master's thesis, Ruhr-Universität Bochum, 2010.

[41] Jinku Li, Xiaomeng Tong, Fengwei Zhang, and Jianfeng Ma. Fine-cfi: fine-grained control-flow integrity for operating system kernels. *IEEE Transactions on Information Forensics and Security*, 13(6):1535–1550, 2018.

[42] Jinku Li, Zhi Wang, Tyler Bletsch, Deepa Srinivasan, Michael Grace, and Xuxian Jiang. Comprehensive and efficient protection of kernel control data. *IEEE Transactions on Information Forensics and Security*, 6(4):1404–1417, 2011.

[43] José Martins, Adriano Tavares, Marco Solieri, Marko Bertogna, and Sandro 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.

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

[45] A. One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[46] Lee Pike, Pat Hickey, Trevor Elliott, Eric Mertens, and Aaron Tomb. Trackos: A security-aware real-time operating system. In *International Conference on Runtime Verification*, pages 302–317. Springer, 2016.

[47] J. Pincus and B. Baker. Beyond stack smashing: recent advances in exploiting buffer overruns. *IEEE Security Privacy*, 2(4):20–27, July 2004.

[48] R. Roemer, E. Buchanan, H. Shacham, and S. Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):2, 2012.

[49] AliAkbar Sadeghi, Salman Niksefat, and Maryam Rostamipour. Pure-call oriented programming (pcop): chaining the gadgets using call instructions. *Journal of Computer Virology and Hacking Techniques*, 14(2):139–156, May 2018.

[50] H. Shacham et al. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM conference on Computer and communications security*, pages 552–561. New York,, 2007.

[51] D. Sullivan, O. Arias, L. Davi, P. Larsen, A. Sadeghi, and Y. Jin. Strategy without tactics: Policy-agnostic hardware-enhanced control-flow integrity. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2016.

[52] Microsoft Support. A detailed description of the Data Execution Prevention (DEP). https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in. [Online; accessed 18-June-2019].

[53] L. Szekeres, M. Payer, T. Wei, and D. Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, May 2013.

[54] 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*, pages 121–141. Springer, 2011.

[55] Tzi-Cker Chiueh and Fu-Hau Hsu. Rad: a compile-time solution to buffer overflow attacks. In *Proceedings 21st International Conference on Distributed Computing Systems*, pages 409–417, April 2001.

[56] Robert J Walls, Nicholas F Brown, Thomas Le Baron, Craig A Shue, Hamed Okhravi, and Bryan 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.

[57] Yutian Yang, Songbo Zhu, Wenbo Shen, Yajin Zhou, Jiadong Sun, and Kui Ren. Arm pointer authentication based forward-edge and backward-edge control flow integrity for kernels. *arXiv preprint arXiv:1912.10666*, 2019.