Proc. of the International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME)
16-18 November 2022, Maldives

Em-RIPE: Runtime Intrusion Prevention Evaluator for ARM Microcontroller Systems

Gianluca ROASCIO
Dept. of Control and Computer Engineering
Politecnico di Torino
Turin, Italy
gianluca.roascio@polito.it

Gabriele SERRA
TECIP Department
Scuola Superiore Sant'Anna
Pisa, Italy
gabriele.serra@santannapisa.it

Vahid EFTEKHARI MOGHADAM
Dept. of Control and Computer Engineering
Politecnico di Torino
Turin, Italy
vahid.eftekhari@polito.it

***Abstract*—Although they have been known for some time, the security implications of buffer overflows (BOF) continue to rouse great attention among software experts in the academic and commercial sectors. Recently, there has been particular interest in discussing how to mitigate risks deriving from BOF on embedded and IoT devices, which have lower computational capabilities given their low-cost and low-power requirements. Although the literature is rich of solutions for these devices as well, authors often fail to quantitatively compare their techniques with related work from a security perspective, and mostly rely on qualitative analysis. Existing evaluator benchmarks (such as the famous RIPE, introduced in 2011) are designed to be used only on general-purpose systems, e.g., with a rich Linux OS and Intel architecture.**

This paper presents Em-RIPE, a prototype evaluation tool written for assessing protections applied to real-time embedded systems, such as microcontrollers equipped with ARM processors. This first version of the tool supports 105 different possible attack combinations, on which the resilience level of the platform under test can be measured. As experimental data, the obtained protection coverage for major compiler-based firmware protections is reported.

***Index Terms*—security, software security, memory safety, control-flow integrity**

I. Introduction

Today, the security of information systems has become of paramount importance due to their increasing diffusion at every level of organizations and, consequently, to the explosion of cyber attack reports and news, which forcibly bring the problem to everyone's attention. Software security is no longer just an issue inherent to servers or personal computers, but touches most of the objects surrounding us, as they now feature smart controllers on board, which drive their functionality and produce data exchanged on what is now called the Internet of Things (IoT). Such end-points, as opposed to desktop and server systems, are characterized by low resource availability, and the need to run software that is (i) directly interfaced with the hardware, with no supervisor or rich OS in between, and (ii) written through languages that allow for hard optimization, such as C or C++. Despite their advantages, the characterization of these languages as memory-unsafe is well known, as well as their inclination to lead to common programming errors, that result in memory corruption vulnerabilities [1]. Among them, prominent is the *buffer overflow* (BOF) class, which still in the latest Common Weakness Enumeration (CWE) report by MITRE Corporation about 2021, leads the rank of the top 25 most dangerous software weaknesses, with a score of 65.93, almost 20 points above the second [2].

Buffer overflow vulnerabilities are the entry point for a multitude of binary attacks, including code injection [3] and code reuse [4]. Since the Morris Worm chronicles in the late 1980s, literature and companies have offered a plethora of solutions to reduce the risk of buffer overflows or mitigate their malicious effects: stack canary [5], stack replicas [6], memory tagging [7] and authentication [8], and control-flow integrity [9] are some of the most famous examples.

From a scientific point of view, all of these solutions can be easily evaluated and made comparable by the classical metrics used for the performance of computing systems extensions, e.g., the average additional time required to run benchmark applications, or the overhead in terms of chip area, or the additional power consumed. However, from the point of view of their security evaluation, scientific consensus has yet to be reached, and very often, proposals are evaluated in a qualitative way, or anyway not justified on the basis of standard experiments for security.

In 2003, Wilander and Kamkar reported that most of the prevention techniques were missing about 50% of the possible attack forms [10], and contextually they presented a tool able to perform a comparative evaluation on runtime buffer overflows. The tool was targeting 20 different attack combinations. The idea was extended and improved in 2011, when a benchmark called *Runtime Intrusion Prevention Evaluator* (RIPE) was published [11]. RIPE features 850 different attacks forms, obtained by combining different target memory sections, library functions, overflow techniques, and attack codes.

Some relevant solutions against buffer overflows existing in the literature have been evaluated using this testbed [12] [13]

978-1-6654-7095-7/22/$31.00

[14], which represents the real first attempt at standardization. However, RIPE is severely limited by the fact that it only supports the `i386` architecture, and relies on the Linux Application Binary Interface (ABI). At present, there is no possibility of using RIPE for architectures other than Intel, or for other operating systems or platforms. Most of all, the benchmark is not usable in the application domain of embedded devices and microcontrollers, which are particularly at risk for the reasons mentioned above. The absence of an objective benchmark to assess the coverage of a solution against BOF for these systems is severe, and is the cause that moved the present work.

This paper presents a first version of Em-RIPE, a tool that adapts the original RIPE philosophy to microcontroller-based real-time systems. Em-RIPE is currently architected as a proof-of-concept for the ARM-based 32-bit microcontrollers, and has a backend relying on FreeRTOS libraries. Em-RIPE now supports coverage evaluation against attacks on 3 different memory sections, 3 types of pointers, 5 injectable *libc* functions, using 3 different attack codes, for a total of 105 possible attack combinations. The evaluator is released under free-use license, and is available for public download via GitHub[1].

The rest of the paper is organized as follows. In Section II, background elements on BOF and the resulting issues are offered. Section III discusses the requirements needed for a benchmark evaluator in the microcontroller domain. In Section IV, the technicalities of the tool and the reasons for the choices made in the design are given. Section V reports results on the measured coverages for classical compiler-based firmware protections, such as stack canary and data execution prevention. Finally, Section VII concludes the paper and charts the way forward for future work.

II. Background

A. Buffer Overflow

In a program, a buffer overflow occurs when a portion of memory allocated by the code to accommodate a given number of bytes (e.g., from an input) turns out to be insufficiently large to contain them all at runtime. The result is that the excess bytes go on to overwrite adjacent portions of memory [15]. This occurs in applications written in weakly typed languages, such as C and C++, which for example have input or array copy functions that only consist in writing values starting from a certain address, with no indication of their nature or quantity (Figure 1). Overflowing a buffer means corrupting subsequent memory locations, i.e., program variables, but also control data. As is well known, computer architectures accommodate on the stack all function data promiscuously, including its return address. A buffer overflow can result in the corruption of the address to return to, resulting in the hijacking of the program control flow.

Fixing ideas on such a vulnerable stack situation, the attacker may choose to hijack the program to the corrupted memory region itself, where it has possibly injected bytes that

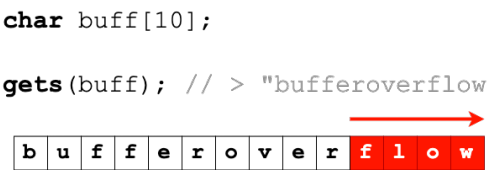

Fig. 1. Graphical representation of a buffer overflow in a C program.

correspond to valid machine code for that architecture. This is the attack technique known as *code injection*. Alternatively, the attacker may inject a sequence of return values subsequent to the first, each pointing to an instruction already in memory, to result in a completely arbitrary chain execution. These attacks are referred to as *code reuse*, and examples include paradigms like Return-into-Libc [16] and Return-Oriented Programming [17].

B. RIPE

As already introduced, RIPE is a testbed published in 2011 [11] that performs a wide range of attacks exploiting buffer overflows for measuring the resilience of the platform on which it runs. It has a back-end written in C and a front-end in Python. The back-end is architected as an application that allocates a set of buffers and then exploits them with code injection or code reuse techniques, using the vulnerable functions of the C standard library as entry points. Through the Python front-end, RIPE allows the selection of 4 different memory sections, 16 target code pointers, 2 overflow techniques, and 10 vulnerable functions, for a total of 850 possible attack combinations. These include 3 different types of shellcode injection, Return-into-Libc, and Return-Oriented Programming. Since it needs to assume the existence of a shell in order to inject it, and since it hosts static binary code for injection or for custom ROP chains, RIPE can only be compiled and executed on Linux x86 platforms. In the paper, the authors provide the results of RIPE runs on different extensions for Linux distributions, including non-executable stack, stack canary protector, LibSafe[2] and StackShield[3]. Since its publication, RIPE has been used on many occasions in the literature to test the effectiveness of protections against BOF implemented at different levels in compatible systems: binary instrumentation [18] [19] [20], custom hardware modifications [21] [22] or employ of legacy hardware features [23] [24] [25].

III. Requirements

The main limitation of the RIPE evaluator is its inability to be compiled and run for systems without a Linux kernel. Although low-power systems can nowadays support light versions of Linux, this does not apply for most of the microcontroller systems (think of monitoring systems in industry, infrastructures, and even home automation). Here,

[1]https://github.com/RHESGroup/embedded-ripe

[2]https://man.cx/libsafe(8)

[3]https://www.angelfire.com/sk/stackshield/

the control software is written directly to run on the bare-metal SoC, in many cases relying on libraries referred to as Real-Time Operating System (RTOS), that are not comparable to a classical operating system, with a consistency on its own. Here, the main aim is just to provide support for deterministic scheduling of different tasks, other than offering libraries for common functionalities. FreeRTOS[4], born from a partnership between market-leading companies in the microcontroller industry, is a fully accessible example. In conclusion, an evaluator for such systems must assume no underlying standalone OS, and must offer test cases that can be run directly on the hardware once compiled.

Similar to what happens with Linux, RIPE is limited by offering support only for machines compatible with an Intel x86-32 architecture. This is rarely used in the embedded domain, where instead the market is dominated by the ARM family. Therefore, the evaluator must make internal use of specific attack codes (e.g., for shellcodes or ROP chains) that are written directly in ARM machine code.

The dual front-end/back-end structure of the original version of RIPE can be instead maintained for its usability and modularity, although the two parts need to be split. The front-end needs to run on an external host system, e.g., a workstation connected to the chip via UART. Through this, it can trigger the execution of the back-end, i.e., the actual body of the evaluator running as the chip software, and evaluate the results. In this way, the Python front-end does not impact the under test system at all. On its own, this latter is not required to have support for Python and multitasking, and can run the evaluator code only.

IV. Features

In light of the above, our Em-RIPE testbed has a front-end written in Python, and a back-end written in C. The front-end runs on a PC host, and consists of a script that iterates over all the different types of supported test attacks. The back-end, on the other hand, executes on the board and consists of an attack generator that builds the attack payload and performs the attack on itself.

Exactly as in the original RIPE, the front-end explores all combinations that can be formed within the *attack space*, and then triggers the software on the board accordingly with the right parameters. The interaction is possible via a connection on the UART port, through which the front-end forwards the execution trigger signal, and in turn receives from the back-end the outcomes of all attack combinations, one after the other.

The dimensions of the attack space are: (i) the overflow location, (ii) the target code pointer, (iii) the vulnerable function, and (iv) the attack code. For each test performed, the result log is marked with one possible outcome: `DONE` when the attack was executed successfully, `NOPE` when the attack encounters an error before running to completion, or `NOT POSSIBLE`, when the attack is not practically possible (e.g., an attack on a return address targeting the BSS section).

[4]https://www.freertos.org/index.html

A. Overflow location

The overflow location describes the memory section in which the target buffer is located. We support attacks on the stack, data, and BSS sections. Em-RIPE does not support the heap section, in order to be as general as possible toward the reference domain: not all microcontroller systems support dynamic memory management, since it requires a non-negligible overhead in terms of speed, consumption and additional components, which is not worth affording for the purposes of the most of control firmwares.

B. Target code pointer

The target code pointer represents the address exploited by the specific attack. It transfers the control flow to the appropriate offset to trigger the payload. Em-RIPE supports attacks to:

- *Function return addresses*: the pushed return addresses on top of the stack, used to jump back to the caller function when the callee terminates;
- *Function pointers*: variables that contains the address of a function, which can be called dynamically;
- *Structures with function pointers*: function pointers which are part of a C structure (`struct`) and lay adjacent to a buffer.

Clearly, function return addresses are stack-specific: this means that they can be used as a target code pointer only when the attack location is the stack. All other targets instead can be allocated in any location.

C. Vulnerable target function

Em-RIPE uses 5 vulnerable functions as attack entry points: `strcpy`, `strncpy`, `memcpy`, `memmove`, and `homebrew`, a loop-based equivalent version of `memcpy` that was originally implemented in the previous version of RIPE.

Such string functions from *libc* allow to copy part of a string from a source buffer to a destination buffer without any boundary check. Then, we use them to overflow the destination buffer. The `n`-versions of the C string functions, such as `strncpy`, instead require the destination buffer size. However, it is up to the developer to provide the destination buffer size. Most of the time, the size is computed dynamically. Therefore, an error in the size computation can frustrate the limit check offered by those functions.

D. Attack code

The attack code represents the payload that an intruder may want to get running through a buffer overflow exploit. The goal here is to trigger a code that merely provides a clear signal that the exploit has taken place with success. Unlike the original version of RIPE, since it cannot leverage the system capabilities offered by a rich OS (such as the `int 0x80` instruction for Linux x86-32), Em-RIPE uses an attack payload that simply relies on a supervisor call (`svc`), which is featured as a machine instruction directly in the ARM ISA. When successfully executed, the `svc` invokes the associated handler, which is responsible for writing to the UART port the identifier

that allows the front-end to register the successful outcome of the attack (`DONE`). If the attack instead fails, the Em-RIPE code does not continue in the direction of the injected code, but through its normal flow, writing a negative identifier (`NOPE`) to the UART port.

Em-RIPE actually supports 3 variants of SVC-based attack code:

- *Injected SVC*: it is a code injection attack in which the payload consists of a single `svc` instruction;
- *Injected SVC with NOP sled*: in the payload, the `svc` is preceded by a number of `nop` (No-Operation) instructions. This second version is actually the one more often used by attackers, as it increases their chances of redirecting the program flow within the injected code. Similarly, this second attack code can be useful in testing countermeasures based on the recognition of common attack patterns [26], such as `nop` sleds are;
- *Reused SVC*: in this last attack, there is no code injection, but the target code pointer is overwritten with that of an existing *gadget* in memory, which contains the `svc` instruction of interest. Such a gadget is artificially inserted inside the back-end code, and cannot be reached in any other way. This attack code is useful for testing the specific resilience of a protection against code reuse attacks, such as Return-into-Libc and Return-Oriented Programming.

In the first two cases, the payload has such a format:

```
(NOP sled +)
attack code +
padding bytes +
address to payload start
```

where the last element must coincide with the location of the pointer to the code to be corrupted. In the last case, instead, what is injected is just the gadget address, plus a padding necessary to align it with the target pointer.

E. The use of FreeRTOS

The Em-RIPE back-end is not offered as standalone firmware, compiled or provided with the entire source, but is actually configured as a FreeRTOS application. This essentially grants abstraction from numerous target-specific low-level details, which could cause the evaluator to lose generality. FreeRTOS is architected in such a way as to support Hardware Abstraction Layer (HAL) services from major ARM-based microcontroller families, including STMicroelectronics, NXP, and FreeScale. Through the public GitHub repository of FreeRTOS[5], it is possible to download demo applications for each of the supported devices, and to integrate Em-RIPE sources into the files for the selected target. The user has to pay for the effort of customizing configuration-specific elements, such as the startup procedure and the linker script for compilation, but in exchange, she gets the benefit of a considerable modularity and portability. Moreover, FreeRTOS can be rather defined as a system of libraries surrounding the custom software to be run on the board, than a full-fledged OS with a life of its own. Hence, the requirement discussed in Section III is preserved, whereby the evaluator must not assume any kind of pre-configured underlying system software that imposes usability constraints.

[5]https://github.com/FreeRTOS/FreeRTOS/tree/main/FreeRTOS/Demo

V. Evaluation

For prototyping, we tested the functionality of Em-RIPE by compiling it with a restricted combination of security options provided by ARM GCC version 9 [27], namely *stack protector* and *data execution prevention*. The evaluator was run on the Netduino-Plus 2 platform, featuring an STM32F405RG microcontroller, which uses an ARM Cortex-M4 core. The board was virtualized using QEMU version 6.

The setups tested are incremental and range from the version with no protections (i.e., with executable data and no stack guard), to one with both guards employed. The data obtained are offered in this Section, preceded by a brief survey of the technicalities of the considered protections.

A. Stack Protector

Using the `--protect_stack` compile option, a guard variable is placed onto the stack between a function's vulnerable local buffers and its stacked return address. Buffers are considered vulnerable in a coarse manner, i.e., when they have automatic storage duration, or are of type `char`. The compiler keeps a global reference copy of the guard value, so it can check whether the value on the stack has remained the same before returning to the caller. If the values are different, the procedure void, a buffer overflow attempt is assumed and the routine `__stack_chk_fail()` is called. The routine content is chosen by the programmer. The value of the guard is contained in a global variable `void* __stack_chk_guard`, which should be provided with a suitable value, such as a strong random number.

B. Data Execution Prevention

Since version 6, ARM introduced the XN (eXecute-Never) bit feature, which prevents execution of bytes from specific memory ranges directly by hardware. Such protection removes the attacker's ability to inject arbitrary code e.g., into the data memory of the application, and then redirect the flow to execute this payload. For our evaluation, we directly set XN options with appropriate flags on the linker script (`.ld` file) provided to the compiler, marking regions that contain data as not executable. In fact, the compiler already supports data execution prevention by default. Therefore, it has been necessary to specify the opposite, i.e., execution from data sections has been specifically allowed when we needed to evaluate a vulnerable setup.

C. Results

Table I shows the results measured on four different combinations of the setups described above, from the most vulnerable to the most protected one. Results are shown in relation to

TABLE I
SUMMARY OF EVALUATION RESULTS USING EM-RIPE'S 105 ATTACK FORMS.

Setup	Overall Effectiveness	Successful Attacks	Failed Attacks
Baseline (no protection)	**0,00%**	105 (100,00%)	0 (0,00%)
Stack guard only	**14,28%**	90 (85,71%)	15 (14,28%)
XN enabled only	**67,67%**	35 (33,33%)	70 (66,67%)
XN enabled + stack guard	**80,95%**	20 (19,05%)	85 (80,95%)

the 105 successful attack combinations: in fact, while having in principle $3 \times 3 \times 5 \times 3 = 135$ possible combinations, the run returns `NOT POSSIBLE` in 30 cases, i.e., when the return address is considered as the target pointer in BSS and data sections, which is known not to be possible. Therefore, while the stack can have all $3 \times 5 \times 3$ combinations, BSS and data can have only $2 \times 5 \times 3$, for a total of 105 possible attacks.

As can be appreciated, the least protection is offered by the application of the stack protector only, which leads to failure in only 14% of the considered attacks. However, significantly higher is the rate of failed attempts as a result of XN bit application only (67%), since code injection attacks are more numerous in the considered attack set. Finally, the application of both measures together leads to 81% of coverage against BOF attacks: the remaining 19% successful attacks represent code-reuse exploits with targets other than the stack, for which more advanced fine-grained CFI mitigations are needed.

The results are in line with what might be expected, and therefore demonstrate the reliability of the work to first outline such an evaluation tool.

VI. RELATED WORK

As already introduced, the security evaluation of solutions against the malicious effects of buffer overflow vulnerabilities does not have a homogeneous path in the literature, and for the most part, qualitative analyses have been presented alongside techniques in order to assess them.

The absence of a quantitative metric in this regard is well denounced in [28]. Here is presented a framework that takes into account the general requirements that a solution against buffer overflow should have, such as scalability and adaptability to legacy systems. The paper produces a report that summarizes the gaps that are identified in 24 selected protection mechanisms.

From an international standard perspective, there is SARD [29], a database maintained by NIST that collects about 170.000 case test programs, containing 150 different buffer overflow vulnerabilities. The test cases are in the form of source code in major languages (C, C++, Java), but also executable files, and include production, synthetic and academic examples. Users can interact with the database through an interface that allows indexed searches by categories, such as vulnerability type, keywords, and size. Being a database, test cases cannot be modified or removed, but entries can be tagged as deprecated and replaced by an updated version.

With regard to the IoT and embedded domain, to the best knowledge of the authors there is no (or no public) ensemble framework that can allow comparable evaluation between solutions against buffer overflows. A recent work by Zhou and Chen [30] presents a version of RIPE adapted for ARM systems, tested for Raspbian OS, the operating system in use on the Raspberry Pi platform[6]. The paper offers data on the system's response once common OS solutions are applied, comparing it to the same as Ubuntu 16.04. However, the work offers no support for platforms that do not have a rich OS and the ability to interface with the Linux ABI and system calls. In addition, the paper does not provide any public resource from which to detect further details about the tool.

Also relevant is the very recent paper by Calatayud *et al.* [31], which offers a comparative analysis of solutions against buffer overflows on high-end IoT devices, tested on different lightweight operating systems such as Xubuntu, Raspbian and Chromium. The analysis is accomplished through a rewrite of RIPE for the 32-bit ARM architecture. However, the outcome is a tool of limited generality, highly customized on the specific analysis accomplished in the study.

VII. CONCLUSIONS AND FUTURE WORK

The paper has presented Em-RIPE, a tool for automatically assessing the resilience of an ARM microcontroller platform to buffer overflow attacks. The tool follows the same philosophy as the original RIPE, combining together different attack payloads, memory sections, target pointers, and vulnerable functions, so to reach a set of 105 possible attack forms to measure the coverage of a security solution. The evaluator has no specific dependencies on a software preset, such as a rich OS, and can be built and run from scratch on the platform to be evaluated. All code is released free of charge, and is available for the community on GitHub.

The discussed release represents a proof of concept, intended to make its own scientific contribution in a domain that still lacks a stable standard for evaluating coverage from BOF attacks. Exactly as the original RIPE, only one payload is provided for each target combination, when when these are in fact theoretically infinite. Therefore, the value of coverage should be meant as an upper bound on the preventive effectiveness for the attack forms considered. Similarly, the use of such a tool *per se* cannot guarantee an objective figure on coverage, since countermeasures could be adopted specifically to counter the attack forms present in this or a later version of the tool. In this regard, we believe in the fairness of researchers who should make use of it.

In addition to a natural extension of the considered attack cases, Em-RIPE needs to be extended and tested on other

[6]https://www.raspberrypi.com/

embedded platforms of the ARM family, which may not have the specificities for which the attack vectors presented apply. Also, the use of facilities for evaluation on multiple builds, to be integrated into the front-end, would be appreciable, for example, for the assessment of protections based on address randomization.

ACKNOWLEDGEMENTS

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] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal War in Memory," in *2013 IEEE Symposium on Security and Privacy*, pp. 48–62, 2013.

[2] C. W. Enumeration, "2021 CWE Top 25 Most Dangerous Software Weaknesses." https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html, 2021. [Online; Accessed May 20th, 2022].

[3] A. One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, pp. 14–16, 1996.

[4] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*, pp. 552–561, 2007.

[5] 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," in *USENIX security symposium*, vol. 98, pp. 63–78, San Antonio, TX, 1998.

[6] N. Burow, X. Zhang, and M. Payer, "SoK: Shining Light on Shadow Stacks," in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 985–999, 2019.

[7] N. Zeldovich, H. Kannan, M. Dalton, and C. Kozyrakis, "Hardware Enforcement of Application Security Policies Using Tagged Memory," in *OSDI*, vol. 8, pp. 225–240, 2008.

[8] R. Elbaz, D. Champagne, C. Gebotys, R. B. Lee, N. Potlapally, and L. Torres, "Hardware mechanisms for memory authentication: A survey of existing techniques and engines," *Transactions on Computational Science IV*, pp. 1–22, 2009.

[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] J. Wilander and M. Kamkar, "A comparison of publicly available tools for dynamic buffer overflow prevention," in *The 10th Network & Distributed System Security Symposium 2003 (NDSS), San Diego, California, USA*, p. 149, Internet Society, 2003.

[11] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "RIPE: Runtime Intrusion Prevention Evaluator," in *In Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC*, ACM, 2011.

[12] M. Khandaker, A. Naser, W. Liu, Z. Wang, Y. Zhou, and Y. Cheng, "Adaptive Call-Site Sensitive Control Flow Integrity," in *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, 2019.

[13] R. K. Shrivastava, K. J. Concessao, and C. Hota, "Code Tamper-Proofing using Dynamic Canaries," in *2019 25th Asia-Pacific Conference on Communications (APCC)*, pp. 238–243, 2019.

[14] M. Castro, M. Costa, and T. Harris, "Securing Software by Enforcing Data-Flow Integrity," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, (USA), p. 147–160, USENIX Association, 2006.

[15] "CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer." https://cwe.mitre.org/data/definitions/119.html, 2022. [Accessed May 22th, 2022].

[16] 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.

[17] 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, pp. 1–34, 2012.

[18] M. Zhang and R. Sekar, "Control flow integrity for {COTS} binaries," in *22nd USENIX Security Symposium (USENIX Security 13)*, pp. 337–352, 2013.

[19] V. Kuznetzov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *The Continuing Arms Race: Code-Reuse Attacks and Defenses*, pp. 81–116, 2018.

[20] J. Li, X. Tong, F. Zhang, and J. Ma, "Fine-cfi: fine-grained control-flow integrity for operating system kernels," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 6, pp. 1535–1550, 2018.

[21] X. Ge, W. Cui, and T. Jaeger, "Griffin: Guarding control flows using intel processor trace," *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 585–598, 2017.

[22] J. Zhang, R. Hou, W. Song, Z. Zhan, B. Zhao, M. Chen, and D. Meng, "Stateful Forward-Edge CFI Enforcement with Intel MPX," in *Conference on Advanced Computer Architecture*, pp. 79–94, Springer, 2018.

[23] S. Das, W. Zhang, and Y. Liu, "A fine-grained control flow integrity approach against runtime memory attacks for embedded systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 11, pp. 3193–3207, 2016.

[24] W. He, S. Das, W. Zhang, and Y. Liu, "No-jump-into-basic-block: Enforce basic block CFI on the fly for real-world binaries," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2017.

[25] J. Zhang, B. Qi, Z. Qin, and G. Qu, "HCIC: Hardware-assisted control-flow integrity checking," *IEEE Internet of Things Journal*, vol. 6, no. 1, pp. 458–471, 2018.

[26] D. J. Day, Z. Zhao, and M. Ma, "Detecting Return-to-libc Buffer Overflow Attacks Using Network Intrusion Detection Systems," in *2010 Fourth International Conference on Digital Society*, pp. 172–177, 2010.

[27] A. Developer, "ARM Compiler for Embedded User Guide Version 6.18." https://developer.arm.com/documentation/100748/0618/, 2022. [Online; Accessed June 10th, 2022].

[28] N. R. Kisore, "A qualitative framework for evaluating buffer overflow protection mechanisms," *International Journal of Information and Computer Security*, vol. 8, no. 3, pp. 272–307, 2016.

[29] P. E. Black *et al.*, "Sard: A software assurance reference dataset," in *Anonymous Cybersecurity Innovation Forum.()*, 2017.

[30] S. Zhou and J. Chen, "Experimental Evaluation of the Defense Capability of ARM-based Systems against Buffer Overflow Attacks in Wireless Networks," in *2020 IEEE 10th International Conference on Electronics Information and Emergency Communication (ICEIEC)*, pp. 375–378, 2020.

[31] B. M. Calatayud and L. Meany, "A comparative analysis of Buffer Overflow vulnerabilities in High-End IoT devices," in *2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)*, pp. 0694–0701, 2022.