A Secure Canary-Based Hardware Approach Against ROP

A Secure Canary-Based Hardware Approach Against ROP

Mahboobe Sadeghipour Roodsari[1,2,*,†], Ebrahim Nouri[3,†], Fatemeh Sheikhshoaei[3,†], Paolo Prinetto[1,2,4,†] and Zainalabedin Navabi[3,†]

[1]*Dipartimento di Automatica e Informatica, Politecnico di Torino, Turin, Italy*
[2]*Cybersecurity National Laboratory, Consorzio Interuniversitario Nazionale per l'Informatica, Italy*
[3]*University of Tehran, Tehran, Iran*
[4]*IMT Scuola Alti Studi Lucca, Lucca, Italy*

Abstract

With the growing demand for embedded systems, from home appliances to industrial usage, security is a significant challenge. Return-oriented programming (ROP) and Code reuse attacks are among the most dangerous attacks. They aim to hijack the control flow program to bypass system restrictions and/or execute malicious code. Stack canary is a well-known defense mechanism against these types of attacks. Generally, they employ a secret value to sit in the memory right before a specific location of the memory that is about to be protected. The security of such a system relies on keeping the canary value secret. However, for a single, unaltered key, it is difficult to guarantee secrecy.

On the other hand, having limited processing capabilities and restricted resources presupposes embedded designers. Security is critical for many real-time applications (like industrial IoT devices), and any protection must comply with the processor speed and memory capacity. This paper proposes a lightweight hardware extension to protect the memory against ROP attacks. The proposed method is a canary-based technique that utilizes Physical Unclonable Functions (PUF) to generate dynamic, unpredictable values. The canary generator security module works in parallel with the processor to avoid any extra performance overhead. In general, our technique is independent of system architecture and even supports processors with multi-execution units.

Keywords

Control flow attack, Return Oriented Programming (ROP), hardware security, stack canary, Physical Unclonable Functions (PUF),

1. Introduction

As embedded systems for IoT devices become prevalent, more serious precautions have to be taken to guarantee such systems' reliability and privacy. Meanwhile, the heterogeneous nature of these systems opens a wide variety of vulnerabilities in the software, especially when using low-level programming languages with numerous controls over hardware resources. The security concern of such systems has been growing in different areas, and security extensions

ITASEC'22: Italian Conference on Cybersecurity, June 20–23, 2022, Rome, Italy
*Corresponding author.
†These authors contributed equally.
mahboobe.sadeghpourroodsari@polito.it (M. S. Roodsari); nouri.ebrahim@ut.ac.ir (E. Nouri); fns.shoaei@ut.ac.ir (F. Sheikhshoaei); paolo.prinetto@polito.it (P. Prinetto); navabi@ut.ac.ir (Z. Navabi)

have been adopted, such as secure boot, secure run-time operations, and memory protections. Software-based attacks have been the subject of research for a long time [1], [2].

This paper focuses on one of the most vulnerable and possible types of software attacks on memory, called *buffer overflow* on the system stack. In processor architectures, the system stack is a portion of the main memory used to load and store functions' local variables, pointers, and return addresses at run-time. By exploiting the software vulnerabilities (e.g., standard library functions), the attacker can access a portion of the legitimately private memory, or hijack the program's control flow for malicious reasons, by directly injecting malicious data and/or manipulating the code pointers. This is what happens in major exploits, such as in Return-Oriented Programming (ROP), where the function return address is corrupted. Based on [3], more than 80% of vulnerabilities can be exploited by control flow attacks. Therefore, a lightweight and secure mechanism is needed to overcome this attack. Numerous techniques have been proposed to make the system resilient to this threat by utilizing software or hardware solutions. Most of the software-based solutions come with different security bugs and a great cost of performance [4], [5]. So, to reduce the performance overhead of software techniques, some hardware approaches have been proposed.

Data Execution Prevention (DEP) is a traditional defense mechanism that splits the memory into two parts: each memory page is either executable or writable, BUT not both ($W \oplus X$). Although many techniques consider the code segment of the memory untouched based on this protection, recent research proves it to still be vulnerable [6].

Address Space Layout Randomization (ASLR) techniques try to muddle regular memory organization, hoping to make it difficult for an adversary to detect addresses [7], [8], [9], [10], [11]. Other than the massive performance overhead due to the randomizing of the address space and moving data among the different memory segments at run-time, these methods have been highly vulnerable to side-channel attacks and the growing field of machine learning [12].

Control Flow Integrity (CFI) techniques aim to verify whether the path taken at run-time follows a set of predetermined paths on the so called Control-Flow Graph (CFG). In general, these methods use a labeling system corresponding to the program's control flow instructions, by matching all possible source and destination pairs. Most CFIs use a shadow stack to store the previous return addresses in an embedded separate secure cache [13], [14], or a dedicated memory cache in the processor [15], [16], [17], [18]. Although this is a secure approach for ROP attacks, the overhead of the needed memory is considerably high. Also, due to the limited capacity of the shadow stack memory, the shadow stack may overflow for programs with highly nested loops or recursions, which makes it vulnerable.

Stack canaries mark the memory with a private keyword placed on the system stack to ensure that specific regions have not been tampered by attackers. Any out-of-bounds memory write changes these values, informing the protection unit about unauthorized activities. A significant advantage of this technique is its negligible overhead in terms of hardware, power, and execution time compared to the other solutions. In the traditional approaches, the private keyword was a fixed value during execution [19], allowing an attacker to reach the private keyword and attack the system. Many dynamic key generation approaches were proposed to overcome the problem [20], [21]. However, most of them lead to an increase in the hardware or execution time overhead.

The present paper proposes a technique for enhancing the stack canary security exploiting

Physical Unclonable Functions (PUF) to improve the security of the canary and keep the main points of it, such as being lightweight and fast. The main contributions of this work can be summarized as follows:

- Using the proposed methodology, the code can be changed dynamically during the run-time. This eliminates the need for pre-analyzing the code;
- The defence mechanism is provided by adding a minimal hardware to the processor. A secure PUF is used for generating the random canary in each function call, in spite of using a table of pre-calculated random numbers;
- The proposed approach is not customized for a specific processor or compiler and can be easily adapted to any processor.

The rest of the paper is organized as follows. Section 2 introduces some backgrounds such as buffer overflow attack, canary, and the PUF. Section 3 provides an overview of the previously proposed techniques using stack canary. The system and attack environment hypotheses and the challenges in canary design are reviewed in Section 4. Section 5 describes the proposed security technique. A preliminary evaluation is shown in Section 6, and finally, conclusions are drawn in Section 7.

2. Background

Control-flow attacks are considered among the most dangerous threats of today's processors. In such an attack, in the absence of enough monitoring, an attacker tries to get the control of the program or read critical data by jumping to any part of the memory. Buffer overflow is one of the most common control-flow attacks on ROP. One of the countermeasures to confront buffer overflow attack is, for instance, stack canary.

The present section offers the necessary background knowledge of the buffer overflow attack, stack canary, and PUF utilized in the proposed canary.

2.1. Buffer Overflow

Stack-based buffer overflow occur when a code is reliant on user data and there is a deliberate misuse of data size to write out of a given memory bound. Having no boundary to write on the system stack, different attacks aim to change particular values such as return address to hijack control of the program. A simple example of such an attack is shown in Figure 1. In this case, the adversary writes a contiguous value starting at one buffer and aims to change the return address to jump to anywhere in the memory and take the program under her control.

2.2. Stack canaries

Stack canary is one of the well-known countermeasures for buffer overflow attacks. A canary is a secret keyword placed on top of the return address in a local function frame, as shown in Figure 2. The canary is pushed onto the stack upon function entry. Considering the canary location, any attempt to change the return address by buffer overflow attack would necessarily affect the canary value as well. Since the canary value will be compared with the stored value

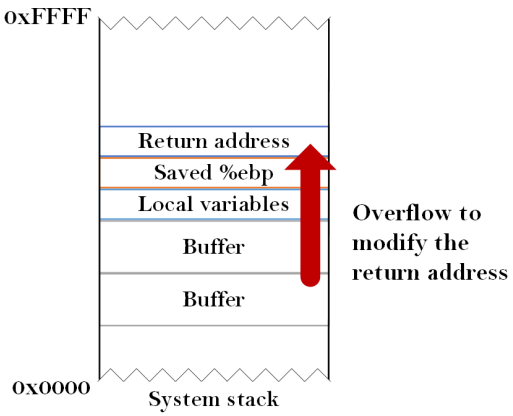

Figure 1: Buffer overflow attack.

in the processor before jumping to the return address, any corruption on the canary value will be detected as an attack. Although a stack canary is widely deployed to secure the system stack from buffer overflow attacks, the security of the return address is highly dependent on concealing the value of the canary from the attacker. If this value is revealed to the attacker, the attacker can easily bypass the canary security. In traditional systems, the canary secret value might have a short bit length, and therefore could be easily brute-forced [22]. Regarding this vulnerability, some works try to generate dynamic values for canary, as discussed in Section 3.

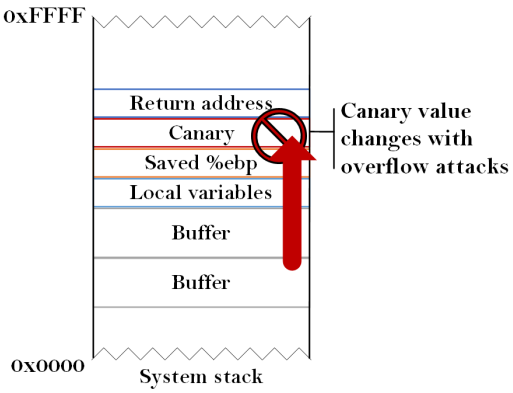

Figure 2: Stack canary mechanism.

2.3. Physical Unclonable Functions (PUF)

As the attention to safety and security grows, the need for authentication and identity validation has become more and more critical. Making use of cryptographic techniques has been a common method to protect data. The drawback of data encryption techniques such as hash functions is

a considerable time and hardware overhead. The exploitation of Physical Unclonable Functions (PUF) is a substitution for the traditional process of generating a secret key and storing it. As opposed to cryptographic algorithms and other hardware obfuscation techniques, PUFs are naturally immune to many attacks due to their unpredictable and random nature. A PUF uses the inherent device process variation to produce a digital signature that is unique and unclonable, yet easy to be reproduced by the device. Usually, a PUF takes an input (challenge) to generate an output (response), which acts as an identifier for that particular input. Since manufacturing process variation is practically always different from one chip to another, the values returned by PUFs act like "fingerprints" for each device. To assess trustworthiness of a PUF, the properties such as *evaluation time*, *uniqueness*, *reproducibility*, *unpredictability*, etc., must be considered [23].

3. Literature Review

Several works have been done to implement canaries in vulnerable systems for detecting buffer overflow attacks. The original method, presented in [19] in 1998, is featuring static canaries. Although this technique is lightweight, it is susceptible to memory leaks, and can be guessed by an attacker in few iterations. Various methods have been proposed based on dynamic canary generation to overcome this drawback. These methods add non-negligible hardware or software overhead to the canary mechanism, thus lowering canary lightness. One can generate a dynamic random canary periodically, and suddenly update all system canary values in the memory [24], but this brings a huge overhead. However, pre-preparing random canaries and storing them in a table is an accepted existing procedure, which is utilized by some works that will be explained in the sequel.

DynaGuard [25] and DiffGuard [26] are compiler-level methods for assigning different canaries to different function frames and changing them after each return. Huang *et al.* [27] propose a method using symbolic canaries, and before forking a call, the canary is checked to see whether it is still one of the symbols or not. Nowadays, an attacker can easily hijack symbols due to the limited number of them.

Some other works generate random numbers using code/system information at run-time. Piromsopa *et al.* [20] add some instructions to generate software-based random numbers for each canary word. To prevent repetition, searching is done through generated random values. This implies a huge and unacceptable run-time overhead to the system, between 500% and 11600% dependant on the running program. Hawkins *et al.* [28] propose DCR, a randomization at run-time without compiler support, by adding the offset of thread local storage (TLS) to the newly calculated random number. Since re-randomizing stack canaries is not completely applied, an attacker can detect the canary words after a few runs. P-SSP [21] rerandomizes the canaries for a new process/thread or for a new function call. It uses a TLS shadow canary, which stores the outputs from re-randomizing the TLS canary. The extensions of this method, raise average time spent by the function prologue and epilogue from 6 to 278-986 CPU cycles which is unacceptable. The technique presented instead in [29] adds some instructions to each function to generate the stack canary at run-time, based on the current execution history, using a sequence of XOR operations. The drawback of this method is run-time expansion by adding

different instructions to each function. PCan is presented in [30], generating canaries based on ARMv8.3- pointer authentication (PA). This method is particular to a specific platform. In PESC [31], the kernel uses the system's performance monitor counter registers to generate random numbers.

In most of the mentioned methods, canary values must be stored somewhere in the program memory — or maybe in a dedicated processor register — to prevent their reuse, so additional storage is needed. Furthermore, in functions with highly nested calls, if each child function gets a new canary value, a linked list is required to record the canary values' history. However, some works protect the inner loops with the same canary as the first loop, in spite of being less secure. These issues make these methods memory-intensive, in contrast to the advantage of canary methods, which are lightweight.

PUFCanary is presented in [32] as a different method, using PUF to generate a key for each function buffer in the system stack. The challenge of this PUF is the value of the canary address in the stack. The PUF response will be XORed by the response of a TRNG to ensure that the canary value is unique across processes. It generates and adds canaries after each buffer to ensure that the overflow will happen to none of them. However, due to the high number of buffers in each function, this method potentially consumes a large memory overhead and run-time check overhead.

In the next Section, a canary method using lightweight hardware is proposed, making the system secure without adding the large memory/run-time overhead.

4. Assumptions and Challenges

Adversary model: In our method, we assume that an attacker is able to inject arbitrary inputs into the system stack. We also consider the attacker to be limited to software manipulation: any hardware fault-injection or signal sniffing that gives the attacker a full control over reading and writing into the processor internal registers is out of the scope of this paper. Also, writing into the memory is not boundless. Therefore, the adversary can only write into the memory segments specified by the application. The attacker can reset the process as many times as she wants, and has the option to perform a brute force attack in order to obtain the canary value.

Architecture model: We assume that our target architecture is a 32-bit machine. Therefore, 32 bits is the dimension of the value used as the canary value.

Canary challenges: One of the canary-based techniques challenges is to ensure the secrecy of the canary value itself. Static canaries rely only on a single value, which potentially can be attained using different kinds of attacks. Therefore, the dynamic canary has become an interesting subject in recent research. Generating dynamic canary faces three main challenges:

- Generate well-distributed random values to be used as the canary;
- Design a practical security module in terms of hardware and power consumption overhead;
- Achieve reasonable execution time for generating and verifying the canary.

5. Proposed Mechanism

As discussed in the previous Section, the existing techniques are relatively expensive due to the run-time and memory overhead, that inevitably increase power consumption, or the methods are not secure enough as the dynamic secret keys for canaries are easy to guess and/or predict. This is not acceptable for many embedded applications (including IoT). To overcome this issue, we propose our PUF-based canary approach, which exploits a lightweight reproducible dynamic canary to fit resource-limited applications. In the following Section, the work is discussed into canary generation and canary check. After that, the complete module is discussed.

5.1. Canary Generation

We use a conventional canary-based memory organization. In order to generate a random canary for each function, we use a PUF module. PUF is able to generate a random response for each challenge, which is reproducible whenever needed. The benefit of using PUF is the uniqueness of the response on each device, making the response value hard to guess for an outer attacker.

To generate various challenges, one way is to use a table of random numbers. However, it consumes extra memory and shows repetitive behavior after a while, making it insecure. In the proposed method, we use the function return address itself as a seed to generate the canary word; notice that the return address points to the next word with respect to the function call instruction (PC + 4). Using return address as a key has two main goals:

1) the value is dynamically selected and unique for each function call during the run-time;
2) it is reproducible, so there is no need to store this value in an extra cache or memory, as we simply choose the return address.

As shown in the Figure 3, starting from a function call, the address of the following 32-bit instruction (PC + 4) is loaded into the CurrAddr (Current Address) register; PUF takes it as a challenge, and 32 bits of the output are selected as a random response. The RetPUF (Return PUF) register stores the PUF response when PUF generates the completion signal. The registered value in the RetPUF register needs to be stored in a location on top of the return address on the system stack. This ensures that different calls result in different canaries even for the same function. In addition, the dynamic generation of the canary makes its value harder to guess and decreases the possibility of reproducing it later.

Our method does not compel designers to use specific types of PUF. Any PUF can be featured as long as two conditions are met. One is that the generated response should be ready at maximum in one clock cycle after the challenge is fed (such as rig oscillator, or arbiter PUFs). Otherwise, it would not be possible to detect and prevent attacks on time. Also, PUF must generate only one unique response for each given challenge, which means for every input challenge, we get a single and specific response as the output.

5.2. Checking Canaries

For completing the method, the canary must be verified. Since the original canary is not stored in any extra memory, in order to validate it, the architecture should be able to regenerate it by

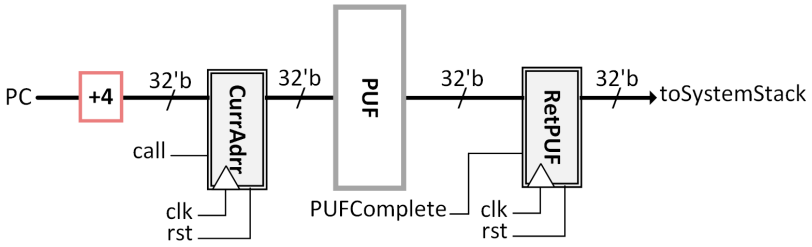

Figure 3: The proposed hardware to generate canaries.

using the return address in the stack. The checking process can be done in parallel with the other tasks on the processor; there is no need to stall the processor and wait for the response. The architecture ensures that the response will be ready before the execution of the next instruction. The overall view of the checking module is shown in the Figure 4. The process is discussed in detail in the following.

When executing the return instruction, first, the canary is loaded, and the 32-bits canary value is stored in StacCan (Stack Canary) register. Then, `PC` goes pointing to the after-call instruction. Its value is again loaded to `CurrAddr` register to re-calculate the canary by using PUF. The result is then compared using the `PUFComp` module (PUF Comparator), and if values do not match, the overflow exception occurs.

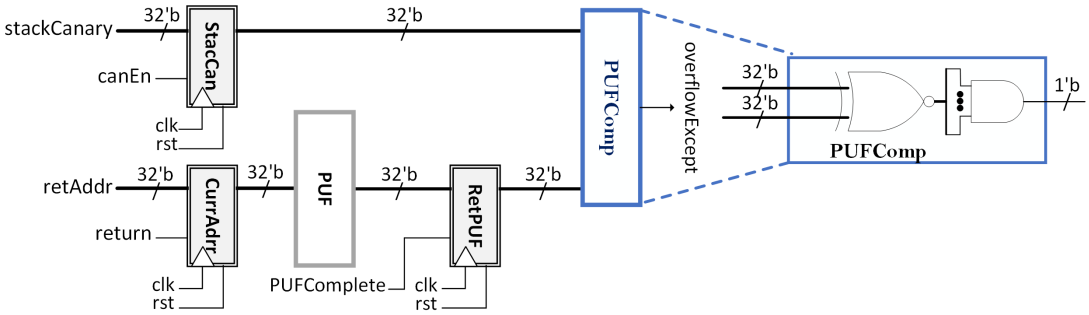

Figure 4: The proposed hardware to check the validity of canaries.

Notably, we share as many components as possible between the two phases. As a result, we only use one PUF module and the same `CurrAddr` register for both generation and checking. The overall architecture is shown in the Figure 5.

The proposed method can be applied to various kinds of applications. It dynamically produces a unique and random canary for each function call, making it extremely difficult for the adversary to guess the value with trial and error. Even in the case of recursion, in which the canary value would be the same, as long as the function is calling itself, the security is still guaranteed. Due to a considerable number of recursive calls in such applications, the canary value appears to be static, which might cause the protection system to fail by revealing the canary. Still, to have a successful attack, one should substitute a malicious address along with its corresponding canary,

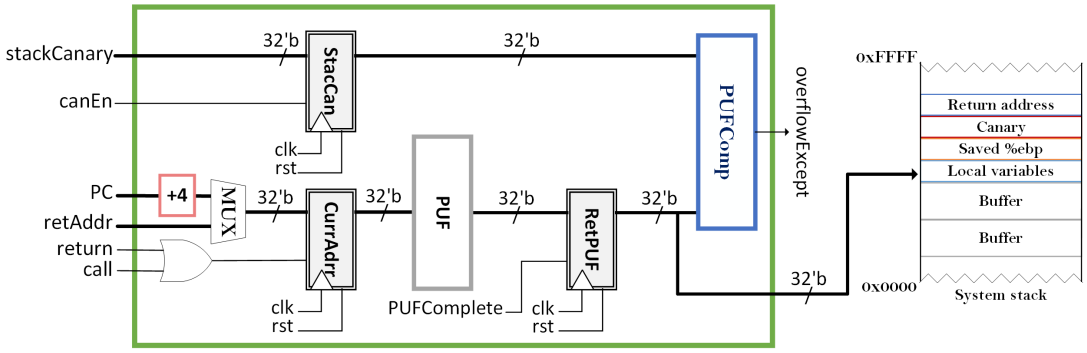

Figure 5: The general view of proposed security module.

which is only possible if the adversary gets access to the system PUF generator. In other words, using this technique, both return address and canary values must be controlled to inject any attack.

The solution just presents a limited risk in the case where the attacker is interested in reusing a return address that has already been used and instrumented with a canary, somewhere else in the code. In that case, through a side attack (e.g., buffer over-read), it could obtain the entire return address-canary pair, and inject it to jump to that destination. However, the attacker's capabilities are extremely limited, and in no practical case this allow her to obtain an arbitrary execution, which is instead coveted by those using this type of attack.

6. Preliminary Evaluation

We evaluate our design by analyzing it against different attacks. Several brute-force attacks are developed that aim to break canary protection. A full brute-force attack tries to guess all possible combinations for the canary value. Therefore, the exploration space is 2^{32}, which might be considered as affordable by modern computation systems [22].

However, our proposed method is immune to these attacks, as realizing canary value alone is not enough to bypass the protection system. We explicitly take two values into account, i.e., the canary value and the return address. To successfully perform the attack, one must produce a corresponding canary for a specific malicious data (targeted return address) that is about to be injected. However, canary generation is only possible through the PUF module, which is closed inside the CPU, and not observable or controllable outside the hardware. Also, in comparison with the other similar canary-based techniques, our proposed method can tolerate discontinuous memory writing, meaning that even keeping the canary location intact does not compromise security. Furthermore, this technique can be applied to the processors with multi-execution units and out-of-order execution.

7. Conclusion and Future Works

Stack canary is an effective, low-overhead technique to detect and overcome stack overflow attacks, but it has its own flaws. Often, a single secret key to protect the return addresses is relied on. Different attacks might reveal this value, making the protection mechanism fail. Using complicated cryptographic systems to secure the secret key makes the overall design remarkably complex. In domains like embedded systems, only a certain amount of hardware overhead is acceptable.

We suggested a lightweight canary-based technique to generate dynamic canaries at run-time. A PUF module is used to make sure no outside attacker is able to reproduce the canary value. Also, no software process has access to generate the canaries. Since the proposed method is not software-based, canary generation and verification can be done parallel with the processor's execution routine, with negligible performance overhead. Furthermore, the technique is not susceptible to brute-force attacks, since for a successful return, a valid canary must always be produced.

Future investigation will involve experimental implementation of the present mechanism into soft cores (e.g., custom RISC-V architectures [33]. Evaluation can be done first with core simulator aided by random number generation to emulate the PUF behavior. This first step could be helpful for assessing security. Then, synthesized version of the instrumented core onto FPGA could be produced to effectively test real PUF and assess overhead in terms of execution times and power consumption.

References

[1] Z. Wang, P. Liu, Position paper: GPT conjecture: understanding the trade-offs between granularity, performance and timeliness in control-flow integrity, Cybersecurity 4 (2021) 1–9.

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

[3] W. He, S. Das, W. Zhang, Y. Liu, BBB-CFI: lightweight CFI approach against code-reuse attacks using basic block information, ACM Transactions on Embedded Computing Systems (TECS) 19 (2020) 1–22.

[4] L. Chen, J. Jiang, D. Zhang, Code reuse prevention through control flow lazily check, in: 2012 IEEE 18th Pacific Rim International Symposium on Dependable Computing, IEEE, 2012, pp. 51–60.

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

[6] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, O. Mutlu, Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors, ACM SIGARCH Computer Architecture News 42 (2014) 361–372.

[7] Y. Liang, X. Ma, D. Wu, X. Tang, D. Gao, G. Peng, C. Jia, H. Zhang, Stack layout random-

ization with minimal rewriting of Android binaries, in: ICISC 2015, Springer, 2015, pp. 229–245.
[8] T. PaX, PaX address space layout randomization (ASLR), http://pax. grsecurity. net/docs/aslr. txt (2003).
[9] J. Ganz, S. Peisert, ASLR: How robust is the randomness?, IEEE, 2017.
[10] S. Bhatkar, D. C. DuVarney, R. Sekar, Address obfuscation: An efficient approach to combat a broad range of memory error exploits, in: 12th USENIX Security Symposium (USENIX Security 03), 2003.
[11] S. Bhatkar, D. C. DuVarney, R. Sekar, Efficient Techniques for Comprehensive Protection from Memory Error Exploits., in: USENIX Security Symposium, volume 10, 2005, pp. 1251398–1251415.
[12] N. Carlini, D. Wagner, {ROP} is still dangerous: Breaking modern defenses, in: 23rd USENIX Security Symposium (USENIX Security 14), 2014, pp. 385–399.
[13] A. De, A. Basu, S. Ghosh, T. Jaeger, FIXER: Flow integrity extensions for embedded RISC-V, in: 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), IEEE, 2019, pp. 348–353.
[14] J.-L. Danger, A. Facon, S. Guilley, K. Heydemann, U. Kühne, A. S. Merabet, M. Timbert, CCFI-cache: A transparent and flexible hardware protection for code and control-flow integrity, in: 2018 21st Euromicro Conference on Digital System Design (DSD), IEEE, 2018, pp. 529–536.
[15] D. B. Roy, M. Alam, S. Bhattacharya, V. Govindan, F. Regazzoni, R. S. Chakraborty, D. Mukhopadhyay, Customized instructions for protection against memory integrity attacks, IEEE Embedded Systems Letters 10 (2018) 91–94.
[16] C. Bresch, A. Michelet, L. Amato, T. Meyer, 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), IEEE, 2017, pp. 57–62.
[17] A. Chaudhari, J. A. Abraham, Effective control flow integrity checks for intrusion detection, in: 2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS), IEEE, 2018, pp. 1–6.
[18] N. Christoulakis, G. Christou, E. Athanasopoulos, S. Ioannidis, HCFI: Hardware-enforced control-flow integrity, in: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, 2016, pp. 38–49.
[19] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, H. Hinton, Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks., in: USENIX security symposium, volume 98, San Antonio, TX, 1998, pp. 63–78.
[20] K. Piromsopa, S. Chiamwongpaet, Secure bit enhanced canary: Hardware enhanced buffer-overflow protection, in: 2008 IFIP International Conference on Network and Parallel Computing, IEEE, 2008, pp. 125–131.
[21] Z. Wang, X. Ding, C. Pang, J. Guo, J. Zhu, B. Mao, To detect stack buffer overflow with polymorphic canaries, in: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), IEEE, 2018, pp. 243–254.
[22] K. T. Dave, Brute-force Attack 'Seeking but Distressing', Int. J. Innov. Eng. Technol. Brute-force 2 (2013) 75–78.
[23] R. Maes, I. Verbauwhede, Physically unclonable functions: A study on the state of the art

and future research directions, Towards Hardware-Intrinsic Security (2010) 3–37.
[24] H. Marco-Gisbert, I. Ripoll, Preventing brute force attacks against stack canary protection on networking servers, in: 2013 IEEE 12th International Symposium on Network Computing and Applications, IEEE, 2013, pp. 243–250.
[25] T. Petsios, V. P. Kemerlis, M. Polychronakis, A. D. Keromytis, Dynaguard: Armoring canary-based protections against brute-force attacks, in: Proceedings of the 31st Annual Computer Security Applications Conference, 2015, pp. 351–360.
[26] J. Zhu, W. Zhou, Z. Wang, D. Mu, B. Mao, Diffguard: Obscuring sensitive information in canary based protections, in: International Conference on Security and Privacy in Communication Systems, Springer, 2017, pp. 738–751.
[27] N. Huang, S. Huang, Z. Deng, Automatic Detection of Stack Overflow Attack in Canary, in: 2018 Eighth International Conference on Instrumentation & Measurement, Computer, Communication and Control (IMCCC), IEEE, 2018, pp. 1418–1423.
[28] W. H. Hawkins, J. D. Hiser, J. W. Davidson, Dynamic canary randomization for improved software security, in: Proceedings of the 11th Annual Cyber and Information Security Research Conference, 2016, pp. 1–7.
[29] R. K. Shrivastava, K. J. Concessao, C. Hota, Code Tamper-Proofing using Dynamic Canaries, in: 2019 25th Asia-Pacific Conference on Communications (APCC), IEEE, 2019, pp. 238–243.
[30] H. Liljestrand, Z. Gauhar, T. Nyman, J.-E. Ekberg, N. Asokan, Protecting the stack with PACed canaries, in: Proceedings of the 4th Workshop on System Software for Trusted Execution, 2019, pp. 1–6.
[31] J. Sun, X. Zhou, W. Shen, Y. Zhou, K. Ren, PESC: A Per System-Call Stack Canary Design for Linux Kernel, in: Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy, 2020, pp. 365–375.
[32] A. De, A. Basu, S. Ghosh, T. Jaeger, Hardware assisted buffer protection mechanisms for embedded RISC-V, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 39 (2020) 4453–4465.
[33] M. Rajabalipanah, M. S. Roodsari, Z. Jahanpeima, G. Roascio, P. Prinetto, Z. Navabi, AFTAB: A RISC-V Implementation with Configurable Gateways for Security, in: 2021 IEEE East-West Design & Test Symposium (EWDTS), IEEE, 2021, pp. 1–6.