X-RIPE: A Modern, Cross-Platform Runtime Intrusion Prevention Evaluator

Gabriele Serra
Scuola Superiore Sant'Anna
Pisa, Italy
gabriele.serra@santannapisa.it

Sandro Di Leonardi
Scuola Superiore Sant'Anna
Pisa, Italy
sandro.dileonardi@santannapisa.it

Alessandro Biondi
Scuola Superiore Sant'Anna
Pisa, Italy
alessandro.biondi@santannapisa.it

***Abstract*—The complexity of modern software systems, the integration of several software components, and the increasing exposure to public networks are making systems more and more susceptible to cyber-attacks. Operating systems and drivers are typically written in C or C++, which are known to be memory-unsafe languages. As a matter of fact, buffer overflows are still a plague in modern software. Despite the consistent amount of research carried out to counteract cyber-attacks, it is still not straightforward to evaluate the worthiness of a specific countermeasure or to identify the appropriate configuration for existing protection tools. In this paper, we present *X-RIPE*, a modern and cross-platform version of the Wilander and Kamkar's RIPE testbed. The objective of X-RIPE is to evaluate how a given countermeasure behaves against buffer overflow threats. We tested X-RIPE against modern memory-corruption protection techniques supported by GCC and Clang/LLVM compilers such as Stack Protector, ASan, and ARM's Pointer Authentication.**

***Index Terms*—security, evaluator, stack-overflow, ripe**

I. Introduction

Software security is a primary requirement for modern systems. Operating systems, especially those developed for embedded systems, are typically written in C or C++. Undoubtedly, these languages offer flexibility and high performance, and, in many cases, they are often the only language supported by the toolchain provided by hardware manufactures for the target platform. However, C and C++ are known to be memory-unsafe languages. Application and operating systems written using memory-unsafe languages could be the target of memory error exploitations [1]. Historically, the adoption of memory protection support mechanisms (MPU) and memory virtualization support mechanisms (MMU) has allowed operating systems to counter several attacks, especially those targeting code injection through memory corruption. Nonetheless, buffer overflows are still present in modern software. Even considering classic buffer overflows only, this class of memory corruption has kept its position on the podium of the *Common Weakness Enumeration* (CWE) SANS top 25 most dangerous software errors for years. In 2022, *Improper Restriction of Operations within the Bounds of a Memory Buffer* laid again at the first place of the CWE SANS ranking [2]. The latest eminent example dates back to January 2022 and was discovered by Qualys Security Advisory [3]. Briefly, they found a buffer-overflow in the C arguments support used by *Polkit* (formerly PolicyKit) that leads to a local privilege escalation from any user to root. Polkit is a component for controlling system-wide privileges in Unix-like operating systems and is installed by default on all major Linux distributions. Interestingly, this recently-discovered vulnerability is technically a memory corruption exploitable since 2009 and has remained latent until 2022. The preceding example demonstrates that buffer overflows and memory-corruption vulnerabilities are still alive and that the induced problem is far from resolved. As a matter of fact, academic and industrial security researchers are still focused on creating countermeasures that can eventually be implemented at the production level. While most of today's defense methods can be seamlessly evaluated from the point of view of run-time overhead, there is no standard benchmark that allows assessing the robustness of a countermeasure. Hence, researchers often resort to qualitative security analysis for a specific technique. In 2003, Wilander and Kamkar [4] developed an elementary tool used to perform a comparative evaluation on run-time buffer overflows targeting 20 different attack combinations. In 2011, Nikiforakis, together with Wilander, leveraged that original idea to develop *Runtime Intrusion Prevention Evaluator* (RIPE), a testbed suite comprising more than 800 buffer overflow combinations. The embryonic tool from 2003 and the subsequent RIPE were used to demonstrate the effectiveness of several tools and techniques, including [5] [6] [7] [8] [9]. RIPE was released under the MIT license in an attempt to standardize the comparison between different countermeasures; however, it targets only the `i386` processor architecture. Practically, there is no benchmark targeting different architectures that allows evaluating countermeasures with the same yardstick and targeting different architectures or environments. In this paper we present X-RIPE, a modern *cross-platform Runtime Intrusion Prevention Evaluator*. X-RIPE is a revamp of the early RIPE project, and it is designed to target multiple processor architectures. X-RIPE already supports `i386`, `x86-64` and `aarch64`. The main objective of X-RIPE is to provide a quantitative evaluation of the protection coverage offered by a specific mechanism against buffer overflows. The tool is released under the GPL license, with the hope to serve as the foundations for a future comprehensive standard penetration test against memory corruption. To test X-RIPE, we applied it against a few modern memory-corruption protection techniques supported by GCC and Clang/LLVM compilers, such as Stack Protector, Address Sanitizer (ASan), and ARM's Pointer Authentication.

Contribution. In summary, this work makes the following contributions:

- It presents X-RIPE, a cross-platform testbed to quantitatively and systematically evaluate the robustness of a buffer-overflow prevention technique.
- It reports on an experimental evaluation that was conducted by applying X-RIPE against anti-memory-corruption techniques supported by the widely-used compilers GCC and Clang/LLVM, specifically: Stack Protector, ASan, and ARM's Pointer Authentication.

Paper structure. The remainder of this paper is organized as follows. Section II reviews the related work. Section III presents an overview of the X-RIPE's architecture. Section IV lists the countermeasures we tested against X-RIPE and their working principles. Section V states results of our experimental evaluation and Section VI concludes the paper.

II. Related work

Software systems are growing in size and complexity, hence the number of bugs. Memory corruption vulnerabilities are among the most frequent potential problems. Dangling pointers, heap meta-data overwrites, uninitialized reads, and invalid or double frees are all examples of these problems. As a result, researchers designed several kinds of protection techniques. Consequently, together with databases of vulnerabilities, testbeds were also developed over the years to measure the effectiveness of those defense techniques.

Among others, SARD (Software Assurance Reference Dataset) [10] is a growing database maintained by NIST (The National Institute of Standards and Technology) of approximately 170 000 programs with a set of known security flaws. These test cases are designs, source code, and binaries from all the phases of the software life cycle: they are mainly written in C, C++, Java, PHP, and C# and cover over 150 vulnerabilities. The dataset includes production, synthetic and academic test cases. The dataset intends to encompass various possible vulnerabilities, languages, platforms, and compilers. Users can view test cases and test suites via the SARD online interface or search for test cases by vulnerability kind, name, size, description words, and other parameters. Many cases include comparable good cases to test for false positives, in which flaws are rectified. In SARD, each test case is described by employing metadata, which encompasses most information regarding the specific flaw or defect. Weaknesses are classified using the Common Weakness Enumeration (CWE) ID and name. The SARD database is archival, which means that once a case is added, it cannot be modified or removed. However, if there are issues with a case, it may be tagged as deprecated and a replacement added. Because most defects are stored in metadata, the findings may be reviewed semi-automatically, displaying the kind of bugs that a tool finds and the false positive rate.

Besides test databases, other research teams tried to standardize test benchmarks for emerging platforms. The leading example is *RIPE-ARM*, an implementation of the RIPE benchmark targeting ARM v7 (32 bit) platforms [11] developed in 2020. In their work, Zhou and Chen performed an experiment using their RIPE-ARM against a Raspberry Pi emulated employing QEMU. Unfortunately, their RIPE-ARM was not publicly released; hence, it has been impossible to take advantage of their implementation. In 2022, Calatayud and Meany worked on a comparative analysis of buffer overflow vulnerabilities in high-end IoT devices. Their analysis still targets 32-bit operating systems [12]. The authors modified the original RIPE by replacing the shellcode for code injection attacks and made that shellcode available to the community. Their work, however, is not a full-fledged benchmark platform, but it is tightly coupled with their analysis; thus, it cannot be generalized. Furthermore, it still targets the ARM v7 architecture. Finally, it is worth mentioning RetTag [13], a hardware-assisted hijacking defense method addressing RISC-V platforms. RetTag leverages the ARM's Pointer Authentication design to enforce pointers' integrity. To perform the security analysis of their technique, they wrote a port of RIPE for RISC-V based platforms. Still, the code is not publicly available.

On the opposite side, several works tried to standardize evaluation methods for defenses from a qualitative point of view. A notable example is [14]. In the mentioned work, the author tried to formalize the general requirements that a protection technique shall implement, such as interoperability with legacy software, scalability, and low-performance overhead. Then, the set of derived requirements was applied to widespread protection techniques. Analyzing 24 various buffer overflow protection strategies using the proposed qualitative methodology, the work outputs a report that summarizes the pros and cons of each of these mechanisms.

III. X-RIPE

X-RIPE is structured as two-layer software, the frontend and the backend. Unlike its predecessor, the backend is independent of the underlying architecture. The internal structure is organized using the *facade* design pattern, thus unifying the architecture-dependent components under a standard API. The backend logic lies on top of a hardware abstraction layer in substance. The X-RIPE backend has an attack generator that builds the attack payload and performs the attack on itself. The backend, indeed, contains vulnerable buffers, gadgets and all the logic to calculate offsets. The X-RIPE benchmark is released under the GPL license and available on Github[1]. The original repository was forked to keep the history of the initial benchmark and incorporate the necessary code to realize the hardware abstraction layer.

A. Tool frontend

The tool frontend is written in Python and consists of a script that iterates all different kinds of attacks. In principle, the frontend allows running all the possible attack forms. The script must be invoked by specifying the available overflow

[1]https://github.com/gabriserra/RIPE

technique among direct, indirect or both. Furthermore, the number of times each attack should be launched and which compiler to target (GCC or Clang) can be specified. The last parameter is optional and is used to control the output format.

```
ripe_tester.py
    <direct|indirect|both>
    <num of repetitions>
    <gcc|clang|both>
    [verbose-options]
```

For each test performed, the result log is marked with one possible outcome: `OK` when the attack was executed successfully, `FAILED` when the attack encounters an error before running to completion, `PARTIAL` when attacks did not succeed in each round, or `NOT POSSIBLE`, when the attack is not practically possible (e.g., a direct attack on a stack buffer targeting a global pointer). The frontend is instructed to explore all the attack space available in the backend.

B. *Tool backend*

The tool backend is written in C and consists of an attack generator. Briefly, it can prepare a malicious payload and perform the attack on itself. The tool is self-contained; namely, it contains both the code to prepare malicious payload and the required vulnerable buffers and gadgets. The attack space has five different dimensions; hence all the possible kinds of attacks generated by the tool are vectors of five components. The exploration of all the attack space combined with the two available techniques (direct and indirect) gives over 2000 various possible attacks.

The dimensions of the attack space are I) the overflow technique, II) the attack code, III) the target code pointer, IV) the memory location, and V) the vulnerable function.

C. *Overflow location*

The attack location describes the memory section in which the target buffer is located. RIPE supports attacks on the stack, heap, data, and BSS sections.

D. *Target function*

There are ten vulnerable functions available as attack vectors: `str(n)cpy`, `str(n)cat`, `s(n)printf`, `memcpy`, `homebrew`, `sscanf`, `fscanf`.

The C library string functions allow copying part of a string from a source buffer to a destination buffer without any control on the destination buffer limit. X-RIPE uses them to overflow the destination buffer. The n-version of the C string functions, such as `strncpy`, instead requires to specify of the destination buffer's size. However, it is up to the developer to provide the destination buffer size. Most of the time, that size is calculated dynamically. Therefore, an error in the size computation can frustrate the limit check offered by those functions. The same is true for functions that require the string format. Indeed, an error in providing the format can let the overflow happens. Concluding, in the list of functions, there is also `homebrew`, a loop-based equivalent version of `memcpy`, implemented originally in the previous version of RIPE.

E. *Attack code*

The attack code represents the kind of shellcode used when the attack occurs. Differently from the original RIPE, our version offers only a shellcode that spawns a shell through `execv` syscall. However, the shellcode is provided in three different flavors.

- Plain shellcode: A shellcode that spawns a shell using `execv` syscall
- Shellcode with NOP sled: The plain shellcode is padded using `NOP` instructions
- Shellcode with polymorphic NOP sled: The basic shellcode is padded using instructions that are equivalent to `NOP` instructions. The polymorphic sled has been generated using the Metasploit framework [15].

Furthermore, X-RIPE offers an additional way of spawning the system shell that does not depend on the provided shellcode avoiding injecting code. The supplementary attack codes started to be carried out after countermeasures such as Data Execution Prevention (DEP) became popular.

- Return-to-libc: Instead of injecting a shellcode, X-RIPE tries to jump to existing libc function, such as `system`
- Return Oriented Programming: X-RIPE uses some instructions already available in the program (gadgets) and combine them to spawn a shell

These last two attack vectors are advanced and more challenging to implement. Therefore, X-RIPE can determine the addresses of libc functions and, additionally, it uses gadgets placed on purpose in the program code.

F. *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 shellcode.

- Previous frame pointer: The address of the frame pointer pushed into the stack, which is used to reference function arguments and local variables.
- Function return address: The return address pointer pushed into the stack is used to jump back to the caller function when the callee terminates
- Longjump buffer: The buffer used to store the current instruction pointer when calling `setjmp`.
- Function pointer: A variable that contains the address of a function callable dynamically.
- Structure with function pointer: A function pointer part of a structure laying adjacent to a buffer.

Clearly, 'Previous frame pointer' and 'Function return address' are stack specific, and 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 each location.

IV. Tested Defenses

The buffer overflow issue became known and was publicly disclosed in early 1972 by the Computer Security Technology Planning Study [16]. The ability to gain the control of a process by overwriting data, however, received worldwide attention thanks to the Morris worm in 1988 [17]. Since then, defending from buffer overflows attacks has been part of security research and several techniques have been developed. For the reasons stated in Section I, C and C++ are notably the most used languages used to develop operating systems and drivers. One aspect that makes those languages flexible is the lack of a (rich) runtime environment. Accordingly, standard versions of C and C++ do not have any memory-bound checking. This design choice is the basis of the portability of the language but, on the other hand, made buffer overflow attacks possible. Consequently, most defense techniques developed over the years can be categorized under two groups, **(i)** mechanisms that introduce bound checking and **(ii)** mechanisms that prevent the consequences the exploitation. Among others, production-ready systems tend to mostly use only those techniques that, during the years, were seamlessly integrated with popular OSes such as Linux and compilers such as GCC or Clang/LLVM. Therefore, we chose to evaluate X-RIPE against the common techniques supported by GCC and Clang/LLVM, namely: Stack Guard, ASan, and ARM's Pointer Authentication. In this section, we are providing a brief analysis of each technique.

A. Stack Protector

At the beginning of the 2000s, Etoh et al. from IBM [18] suggested a modification to the GCC compiler to protect against stack overflows. The main idea was to place a randomly-generated integer, named `canary`, between any stack-allocated buffers and the return address saved on the stack. The name 'stack canaries' is due to their analogy to coal mine canaries, given that they are used to detect whether it is safe to carry on the program execution. GCC Stack Protector inserts stack canaries on the stack of certain functions and is still used today due to its simplicity and the low overhead introduced at runtime. As illustrated in Figure 1, the canary is placed right after local variables in the current implementation, protecting both the old base pointer and the return addresses from direct overflows. Furthermore, the mechanism arranges the local stack variables to ensure that char buffers are always allocated next to the canary. The latter assumption prevents a direct overflow to corrupt any other local variable.

B. AddressSanitizer (ASan)

The *AddressSanitizer* (also known as ASan) is an open-source memory error detector originally introduced by Serebryany et al. from Google [19]. ASan works as a compiler instrumentation module and is currently implemented in Clang (starting from version 3.1 [20]) and GCC (starting from version 4.8 [21]). ASan targets the most common architectures, including x86 and ARM (both 32- and 64-bit versions of architectures). The tool consists of a compiler pass and the

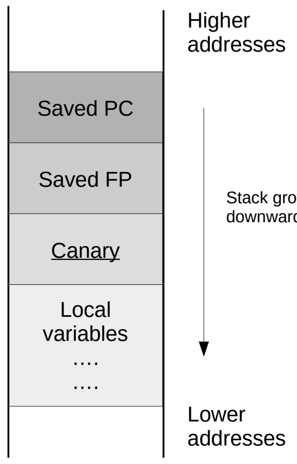

Fig. 1. Conventional layout of a stack frame when stack protector is enabled.

related runtime library. It was designed to find and catch memory errors such as use after free, heap/stack/bss overflows, use after return or scope, etc.

The basic idea of ASan is to divide the virtual address space into two disjoint classes, the main application memory `Mem` and the shadow memory `Shadow`. The regular application code uses the main application memory. On the other hand, the shadow memory consists of a memory area hidden from the application and used to record information about the main memory. The shadow memory contains the shadow values, namely a set of shadow bytes. Shadow bytes, indeed, are mapped to one or more bytes in the main memory. ASan maps 8 bytes of the application memory into 1 byte of the shadow memory. Therefore, the two classes of memory have a correspondence that is built in such a way that computing the shadow memory mapping `mem_to_shadow` is fast. ASan also introduced the idea of poisoned bytes. Poisoned bytes (or *redzones*) are memory areas that cannot be referenced. ASan runtime library can detect accesses to redzones; hence the wider the redzone, the larger the overflows or underflows that will be detected. Poisoning a byte of the memory will result in a particular value written into the corresponding shadow memory.

During the compiler pass, functions such as `malloc` and `free` are replaced with a customized implementation that allocates extra poisoned bytes around the allocated memory region. Furthermore, each memory access that involves a reference to the pointer is transformed. The memory around the area accessed is poisoned too. To make a simple example, suppose that the program accesses a pointer as follows:

```
*address = ...;  // or: ... = *address;
```

If the same program is instrumented by means of ASan, the compiled code would result in:

```
shadow_addr = mem_to_shadow(address);

if (shadow_is_poisoned(shadow_addr))
{
    reportError(address);
}

*address = ...;  // or: ... = *address;
```

That instrumentation causes a runtime error report when the accessed address is not legal.

C. *ARM's Pointer Authentication*

The relevance of security in modern operating systems pushed chip designers to introduce several security-related hardware facilities in their processors. A relevant example is the feature included by ARM in version 8.3 of their ARMv8 processor architecture named Pointer Authentication (PA). Another relevant example is the ARM's Branch Target Indicator (BTI) feature, introduced since version 8.4 of ARMv8. ARM is not the only manufacturer that has invested in this direction. In recent years, Intel has proposed a similar architectural extension called Control-flow Enforcement (CET). The facilities introduced by ARM and Intel can be used to realize robust control-flow integrity enforcement. Almost all CFI techniques watch over a program execution to ensure that the target of an indirect branch is the intended instruction. Among others, they verify that the control comes back to the calling function at each function. Since control-flow hijacking is essential in many exploits (e.g., including those based on buffer overflows), independently of the exploited vulnerability [22], CFI techniques proved to be effective against several widespread attacks and are considered among the most advanced security countermeasures. In a nutshell, ARM's PA works by cryptographically authenticating the content of a register before using it. Indeed, it is conceived as a protection against modification of code pointers such as return addresses stored in memory. For instance, PA represents a valuable protection mechanism to ensure that functions only return to legal locations as expected by the program according to the CFG, hence preventing stack overflow attacks. The BTI mechanism can secure indirect branches, enforcing that the destination location of the branch contains only instructions of an acceptable list. Combining both instruments allows for a complete forward-backwards control-flow integrity, reducing the possibility of an attacker hijacking the execution flow to execute arbitrary code.

V. Evaluation results

This section presents an evaluation campaign performed on modern compiler-supported techniques. The evaluation campaign was carried out with mainly two purposes. The first objective was to understand if our X-RIPE implementation could provide some working attack on a modern OS. Then, the second objective was to understand how the latest protection techniques available on the market, such as ARM's Pointer Authentication, behave compared to well-established methods. Production-ready systems commonly use protection schemes integrated with popular OSes and compilers such as GCC. Therefore, to test X-RIPE, we applied it against memory-corruption protection techniques supported by GCC and Clang/LLVM compilers, namely: Stack Protector, ASan, and ARM's Pointer Authentication presented in the Section IV. Our evaluation was made using Ubuntu 20.04, the long-term support Ubuntu distribution released in April 2020. The distribution was equipped with version 5.13 of the vanilla kernel, compiled with the support for Pointer Authentication, hence enabling the `CONFIG_ARM64_PTR_AUTH` flag in the configuration. ARM's Pointer Authentication support is available only for processors adopting the ARMv8.3-A architecture version (and above). At the time of writing, there are no COTS development boards available on the market today. Therefore, our evaluation campaign was performed using QEMU v6, enabling the TCG (full-software) emulation of the `FEAT_PAuth` architectural feature. The evaluation campaign does not consider time performance; hence, using a virtual machine does not influence our results. To evaluate the protection degree of each specific technique, X-RIPE is compiled, by default, without stack protector (`-fno-stack-protector`) and with executable stack (`-z execstack`). In the following subsections, the results obtained with each technique are analyzed. The results are summarized in **Table 1**.

A. *Stack protector: results*

The stack protector mechanism, both in the version offered by GCC and Clang, is focused on protecting the stack. Stack canaries provide detection of a stack buffer overflow before dangerous code is executed. Results show that both the implementation provided by GCC and Clang/LLVM successfully prevent each kind of stack overflow. Thanks to local variable re-ordering, stack protector works in preventing both direct and indirect attacks. On the order hand, the majority of attacks targeting BSS, heap or data segment are not counteracted.

B. *ASan: results*

ASan is a runtime address sanitizer designed to detect memory errors. Being composed of both compiler instrumentation and a runtime library, ASan is a powerful technique. ASan was intended to prevent out-of-bounds access to the heap, stack, and global objects. Our results show that ASan can detect the most significant part of the overflows, direct and indirect, targeting BSS, data segment, stack and heap. However, some overflows are still not detected, and they are those regarding generic function pointers laying in structures. When a buffer in a structure adjacent to a generic function pointer is overflown, the pointer can be corrupted, replacing its content with, for instance, an address of a different function. Regarding ASan, adjacent buffers in structures are not protected from overflow to avoid backward compatibility issues. It is common, especially in the oldest version of C, to use a structure as an

TABLE I
OVERALL EFFECTIVENESS OF PROTECTION TECHNIQUES FOR BUFFER OVERFLOWS

Setup	**Overall effectivness**	Successful attacks	Failed attacks
Ubuntu 20.04 (GCC, no protection)	62%	1084	1770
Ubuntu 20.04 (Clang, no protection)	61%	1109	1745
Stack protector (GCC)	86%	409	2445
Stack protector (Clang)	84%	470	2384
ASan (GCC)	98%	49	2805
ASan (Clang)	98%	49	2805
Pointer Authentication (GCC)	83%	472	2382
Pointer Authentication (Clang)	83%	477	2377
All protections (GCC)	99%	28	2854
All protections (Clang)	99%	28	2854

array of bytes, independently of declared types of structure members. Furthermore, using ASan to protect a program does not come for free; the typical slowdown introduced by AddressSanitizer is 2x [23].

C. ARM's Pointer Authentication: results

ARM's Pointer Authentication is an architectural feature that comprises a set of instructions and facilities offered by the processor to sign and authenticate pointers. That architectural feature, referred to as `FEAT_PAuth` in Linux, was introduced in 2017 with ARM v8.3-A architecture. ARM's Pointer Authentication support can be used to sign and authenticate any pointer. However, given the relatively recent introduction, the support offered by GCC and Clang/LLVM it is still unripe. So far, Pointer Authentication is used only to authenticate the return address before taking the return branch to the caller. Our results show that the current support is comparable to stack protector in terms of effectiveness; thus, the community's effort must focus on improving the current support available for this outstanding instrument.

D. All protections

When using all protections together, the number of attacks performed with success is nearly null. However, it is still different from zero. In addition, the overhead introduced by that techniques is non-negligible, and, often, not all protection schemes are put in place together. That means all the current protection techniques are still not perfect, and a buffer overflow can still attack a modern OS.

VI. CONCLUSIONS & FUTURE DIRECTIONS

Buffer overflows represent a security threat for applications and operating systems, especially for the embedded systems, where the usage of C and C++ is a common choice. Since the '90s, many countermeasures have been designed and implemented. However, recent exploited vulnerabilities suggest that deliberate memory corruption is still an open problem. In this article, we presented X-RIPE, a cross-platform Runtime Intrusion Prevention Evaluator designed to evaluate, in a quantitative manner, protection coverage offered by a specific mechanism against buffer overflows. We analyzed modern compiler-supported protections against X-RIPE to compare their coverage. X-RIPE attempts to be the basis for a standard way to evaluate a defense technique's robustness; however, it is currently more of a proof-of-concept. In the future, we are willing to extend this work by providing even more, attack vectors to execute several reasonable real-world attacks, which will help quantify the coverage of a specific technique.

REFERENCES

[1] V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos, "Memory Errors: The Past, the Present, and the Future," in *Research in Attacks, Intrusions, and Defenses*, D. Balzarotti, S. J. Stolfo, and M. Cova, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 86–106.

[2] S. Institute, "Cwe/sans top 25 most dangerous software errors," 2022. [Online]. Available: https://www.sans.org/top25-software-errors/

[3] Q. S. Advisory, "pwnkit: Local Privilege Escalation in polkit's pkexec (CVE-2021-4034)," 2022. [Online]. Available: https://seclists.org/oss-sec/2022/q1/80

[4] 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.

[5] 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.

[6] R. K. Shrivastava, K. J. Concessao, and C. Hota, "Code tamper-proofing using dynamic canaries," in *2019 25th Asia-Pacific Conference on Communications (APCC)*, 2019, pp. 238–243.

[7] 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*, ser. OSDI '06. USA: USENIX Association, 2006, p. 147–160.

[8] O. Ruwase and M. Lam, "A practical dynamic buffer overflow detector," in *In Proceedings of The 11th Annual Network and Distributed System Security Symposium*, 2004.

[9] N. Tuck, B. Calder, and G. Varghese, "Hardware and binary modification support for code pointer protection from buffer overflow," in *37th International Symposium on Microarchitecture (MICRO-37'04)*, 2004, pp. 209–220.

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

[11] 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)*, 2020, pp. 375–378.

[12] 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)*, 2022, pp. 0694–0701.

[13] Y. Wang, J. Wu, T. Yue, Z. Ning, and F. Zhang, "Rettag: Hardware-assisted return address integrity on risc-v," in *Proceedings of the 15th European Workshop on Systems Security*, ser. EuroSec '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 50–56. [Online]. Available: https://doi.org/10.1145/3517208.3523758

[14] N. R. Kisore, “A qualitative framework for evaluating buffer overflow protection mechanisms,” *Int. J. Inf. Comput. Secur.*, vol. 8, no. 3, p. 272–307, jan 2016. [Online]. Available: https://doi.org/10.1504/IJICS.2016.079187

[15] Rapid7, “Metasploit: Penetration Testing Software.” [Online]. Available: https://www.metasploit.com/

[16] J. P. Anderson, “Computer security technology planning study,” U.S. Air Force Electronic Systems Division Tech., Tech. Rep., 1972.

[17] E. H. Spafford, “The internet worm program: An analysis,” *SIGCOMM Comput. Commun. Rev.*, vol. 19, no. 1, p. 17–57, jan 1989. [Online]. Available: https://doi.org/10.1145/66093.66095

[18] H. Etoh, “GCC extension for protecting applications from stack-smashing attacks,” IBM Research Group, Tech. Rep., 01 2004.

[19] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, “AddressSanitizer: A fast address sanity checker,” in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, Jun. 2012, pp. 309–318. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[20] L. Team, “LLVM 3.1 Release Notes,” 2012. [Online]. Available: https://releases.llvm.org/3.1/docs/ReleaseNotes.html

[21] G. Team, “GCC 4.8 Release Changes,” 2014. [Online]. Available: https://gcc.gnu.org/gcc-4.8/changes.html

[22] J. Pincus and B. Baker, “Beyond stack smashing: recent advances in exploiting buffer overruns,” *IEEE Security Privacy*, vol. 2, no. 4, pp. 20–27, 2004.

[23] L. Team, “Clang 15.0.0 documentation: AddressSanitizer.” [Online]. Available: https://clang.llvm.org/docs/AddressSanitizer.html