Memory Integrity Techniques for Memory-Unsafe Languages: A Survey

IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 8 February 2024, accepted 11 March 2024, date of publication 21 March 2024, date of current version 27 March 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3380478

SURVEY

Memory Integrity Techniques for Memory-Unsafe Languages: A Survey

VAHID EFTEKHARI MOGHADAM[1], GABRIELE SERRA[2], (Member, IEEE), FEDERICO AROMOLO[2], GIORGIO BUTTAZZO[2], (Member, IEEE), AND PAOLO PRINETTO[1], (Senior Member, IEEE)

[1]Department of Control and Computer Engineering, Politecnico di Torino, 10129 Turin, Italy
[2]TeCIP Institute, Scuola Superiore Sant'Anna, 56124 Pisa, Italy

Corresponding author: Vahid Eftekhari Moghadam (vahid.eftekhari@polito.it)

The Ph.D. research program of TIM S.p.A. (Italy) partially supports the work described in this paper. This work was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union–NextGenerationEU.

ABSTRACT The complexity of modern software systems, the integration of several software components, and the increasing exposure to public networks make systems more susceptible to cyber-attacks, especially those targeting memory. Memory error exploitation received worldwide attention thanks to the Morris worm in 1988 and has been around for over 30 years. As a matter of fact, attacks that involve memory safety, such as buffer overflows, are still a plague in modern software. The research in countering those kinds of attacks has gone in several directions. This work surveys memory integrity techniques developed during the last quarter century for embedded or general-purpose open-source operating systems, ranging from older mechanisms to state-of-the-art solutions. A comparison of various memory integrity techniques is presented to examine their effectiveness and technical significance. Insights into ongoing trends and developments are also provided to assess their potential impact in the foreseeable future.

INDEX TERMS Memory overflows, memory safety, security, unsafe languages, memory safety techniques, security techniques comparison, memory integrity.

I. INTRODUCTION

Software security is today a primary requirement for computer systems and no longer an issue that is only inherent to servers or personal computers. Nowadays, embedded computing systems are employed in diverse application domains to control safety-critical cyber-physical systems including automotive, railway, and avionics systems, nuclear power plants, air traffic control systems, autonomous robots, and military devices, thus playing an extremely important role in our society. The typical software controlling these systems is continuously growing in both size and complexity, thus creating a larger and larger attack surface due to the inevitable introduction of subtle software vulnerabilities. In addition, access to the public network increases their vulnerability, which can be exploited to accomplish malicious actions and cyber-crimes.

Operating systems, from those developed for server computing to those conceived for embedded systems, are typically written in low-level languages, such as 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 manufacturers for a specific target platform. However, those languages are known to be *memory-unsafe*. A memory-unsafe language is a programming language that lacks features or constraints designed to prevent common programming errors related to the freedom in accessing memory. In memory-unsafe languages, programmers have more direct control over the memory, but this

The associate editor coordinating the review of this manuscript and approving it for publication was Rakesh Matam.

freedom can lead to unintended consequences and side-effects, such as memory leaks, buffer overflows, and other vulnerabilities.

Indeed, applications and operating systems written using memory-unsafe languages could be the target of memory error exploits [1]. As a matter of fact, memory corruption vulnerabilities are among the most frequent potential problems. Dangling pointers, heap meta-data overwrites, uninitialized reads, and invalid or double-free vulnerabilities are common examples of such problems. Consequently, security researchers and system designers developed various protection techniques to address these concerns.

The adoption of hardware memory protection and virtualization support mechanisms allow operating systems to counter several attacks. Nonetheless, exploitable vulnerabilities related to memory management are still present in modern software. Even considering classic buffer overflows only, this class of memory corruption vulnerability has kept its position on the podium of the *Common Weakness Enumeration* (CWE) SANS top 25 most dangerous software errors for years. A recent eminent example of buffer overflow exploit enabling unrestricted privilege escalation in Linux-based operating systems dates back to January 2022 [2], when Qualys Security Advisory identified a buffer overflow in the management of C arguments in *Polkit* (formerly PolicyKit), a software component controlling system-wide user privileges, leading to a local privilege escalation from any user to root privileges. Interestingly, this recently discovered vulnerability is technically a memory corruption that has been potentially exploitable since 2009 but has remained latent until its discovery 2022. This example demonstrates that memory corruption vulnerabilities still represent a very serious threat in computer security, and that the related technical and scientific problem of finding adequate countermeasures is far from resolved. Academic and industrial security researchers are still focused on designing countermeasures that can eventually be implemented. However, despite the huge amount of research effort spent in the field, only a few defensive techniques have been actually implemented at the production level, especially due to performance reasons.

In this work, we survey memory safety techniques that have been integrated into production-level systems, ranging from older mechanisms to state-of-the-art solutions. We aim at gathering and analyzing a wide range of techniques that have been proposed and implemented to enhance memory safety, thus providing a comprehensive overview of the advancements in the field. Furthermore, we present a comparative analysis that can aid in evaluating the potential of these techniques for future applications, also identifying areas where further research and improvements are needed.

Contributions: This work provides the following contributions:

- It presents a comprehensive survey of memory safety techniques developed over the past twenty-five years.
- It performs a comparative analysis of the different memory safety techniques identified in the survey, providing valuable insights into the trends and developments within the field.

Survey scope: In examining the multitude of available options, the survey focuses on the most significant techniques employed over the past two decades that have been implemented, or are currently in production, by one or more open-source operating systems and that are supported by one or more major compiler toolchains (i.e., GCC or Clang). The selection criteria prioritize techniques implemented and adopted in the Linux kernel, which stands out due to its widespread adoption for both embedded and general-purpose applications. In Section III, the set of requirements used to center the scope of the survey are explicitly listed.

Paper structure: The remainder of this paper is organized as follows. Section II introduces the necessary background to properly understand the presented methods. Section III sets out a taxonomy for existing techniques and provides a concise explanation of the working principle behind each technique. Section IV presents a comparison of the surveyed techniques considering different evaluation perspectives. Section V shows the limitations of the considered techniques and provides a concrete road map for memory safety in the context of future research. Section VI concludes the paper.

II. BACKGROUND

Memory errors have been investigated since the 1970s, and new memory-related vulnerabilities are discovered ever year. This section presents an overview on the history of memory errors and vulnerabilities and introduces the necessary background concepts.

A. A BRIEF HISTORY OF MEMORY INTEGRITY

Historically, memory errors were first publicly discussed in the 1970s. Specifically, the idea of reading/writing outside the allowed boundaries of a buffer became known and was publicly disclosed in early 1972 by James Anderson in the pivotal Computer Security Technology Planning Study [3]. The ability to gain control of a process by overwriting data received worldwide attention thanks to the Morris worm in 1988 [4]. Since then, *buffer overflows* have been widely recognized as the most well-known exploitation technique in computer security history. In response to the Morris worm exploitation, DARPA founded the Computer Emergency Response Center (CERT). The main goal of CERT is to collect reports about vulnerabilities discovered by users and forward them to software/hardware vendors. Subsequently, numerous mailing lists and public archives of vulnerabilities were created, such as *Bugtraq* and *Full Disclosure* [1], [5], [6].

Nevertheless, until 1995, memory error countermeasures were not heavily researched and discussed. In 1996, Elias Levy published a blog post on Phrack Magazine [7] describing thoroughly how to take advantage of stack smashing. Since then, discussions on protection mechanisms have proliferated, and defending from memory-targeted attacks has been part of security research, with several techniques developed over the years.

B. THE ROLE OF COMPILERS AND LIBRARIES

Software vulnerabilities frequently arise due to either improper checking of input parameters or the presence of unexpected input values. Therefore, the presence of vulnerabilities in a program is strongly related to the software development process, which is itself heavily dependent on the selected development tools. Given a programming language of choice, the key components of a toolchain, such as assemblers, compilers, and debuggers, play a vital role in the development and execution of software programs. Following their significant impact on achieving a high-quality and/or valid result, a software developer should thoroughly understand the behavior of such tools, their capabilities, and their shortcomings, especially when developing software in the security and/or the safety domains [8]. In the following, we provide an overview of necessary background concepts related to the components of the compilation toolchain.

Machine-independent optimizations occur in the optimizer stage of the compilation process. These generic optimization techniques are common among programming languages and are the common place where some unwanted behaviors can be introduced to the programs. Optimizations like reachability analysis and dead-code elimination, constant propagation, or code relocation based on context are some examples of these transformations. At the end of this stage, the code will be passed for further analysis and transformations to the backend phase, in which specific features of the instruction set architecture are exploited.

Indeed, compiler implementations provide their users with many options to allow better tuning of the parameters, such as those targeting memory safety. On the other hand, parameters driving different levels of optimizations (e.g., to improve program performance or to reduce code size when adhering to given memory requirements) could introduce vulnerabilities when used carelessly.

Unwanted alterations and optimization could affect the structure of a program, potentially introducing unwanted side effects. This aspect is particularly critical when data security and integrity are among the main design goals [9], [10]. For instance, one such compiler optimization is dead store elimination (DSE), which removes data store operations into memory locations that are not read by any subsequent instruction. This feature can potentially introduce security vulnerabilities into a program. For instance, consider the following pseudo-code, which performs encryption of some data using an encryption key which is later explicitly overwritten in memory by a sequence of zeros using a memory write operation:

```
uint* encrypt_key = malloc(KEY_SIZE);

    // ... execute encryption algorithm

memset(encrypt_key, 0, KEY_SIZE);

free(encrypt_key);
```

Since the allocated memory buffer, `encrypt_key`, is not used after the call to the `memset` function, the compiler can consider the set memory operation redundant and eliminate it for when applying DSE optimizations. However, if the buffer contains sensitive data, the optimization will cause the data to be left in memory. Attackers can exploit such memory vulnerabilities to disclose information and get access to secret data, which can be leveraged to further compromise the system. Although this compiler behavior is well-known to many developers, it is still a relevant issue when security is a crucial requirement. The developer should be aware of any kind of potentially dangerous optimizations which could jeopardize the security of the application, and consider either disabling those optimizations altogether or applying workarounds to preserve the security of the software [11], [12].

Another important aspect to be considered to improve functional safety and security is software libraries. Commonly used functionalities are generally provided by the development toolchain in the form of libraries to be included in the design flow. Third-party libraries are often subject to targeted security exploits in the software development as well (e.g., cryptographic libraries like OpenSSL[1]). As with any software, the functionalities provided by these libraries could potentially present flaws in their design, and not properly addressing such flaws can introduce vulnerabilities to the programs utilizing a third-party library [13], [14]. A few examples of memory techniques targeting common libraries (such as *glibc*) are presented in the next sections.

C. COMMON ATTACKS

Since the release of the Morris worm, numerous memory vulnerabilities have been discovered and often exploited thanks to carefully crafted attack techniques. The following section briefly presents some of the most common attack patterns employed by such attacks.

1) CONTROL-FLOW HIJACKING

Control-flow hijacking is a common technique used in many exploits, irrespective of the specific vulnerability being exploited. The control-flow graph (CFG) of a program

[1] https://www.openssl.org

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

represents the valid sequence of control transfers within the program, and is represented as a directed graph where nodes represent routines or basic blocks, while edges represent control transfer instructions such as branches, function calls, and returns. The CFG can be constructed offline by analyzing the source code or binary executable, or it can be dynamically discovered during program execution.

Control-flow hijacking attacks try to divert the legal execution path of a program, for instance, by modifying the target of an indirect branch instruction (which jumps to a value computed at runtime) or by forcing a function to return to an address that differs from the expected return address within the calling function.

Control-Flow Integrity (CFI) is a set of security measures aimed at ensuring that the execution flow of a program follows the intended paths defined by its canonical CFG [15]. CFI techniques focus on monitoring the execution of a program to ensure that control transfers adhere to the CFG, and have been proven to be effective against many well-known attacks and are considered advanced security countermeasures [16].

2) BUFFER OVERFLOWS

A buffer overflow (or overrun) is an anomaly occurring when a portion of memory allocated to store a given number of bytes is insufficient to contain a larger-sized payload. Therefore, the excessive bytes are written to adjacent portions of memory [17]. This anomaly often occurs when applications are written in languages such as C and C++, which, for instance, have input functions or array copy functions that only consist in writing values starting from a certain address, with no explicit limit on the amount of memory to be copied (e.g., refer to Figure 1). Standard versions of C and C++ do not have any memory-bound checking. This design choice enhanced the portability of the language but, on the other hand, made overruns possible.

Overwriting memory locations adjacent to a buffer corrupts program variables, which can contain control data (e.g., pointers to functions or stored program addresses). Therefore, when a buffer overflow occurs, it can lead to the manipulation and corruption of the intended address to be jumped to, effectively hijacking the program's control flow.

Furthermore, by leveraging buffer overflow, the attacker may hijack the program to execute code stored in the corrupted memory region itself, where it has previously injected bytes corresponding to valid machine code for that architecture (e.g., with the same memory copy operation used to trigger the buffer overflow vulnerability). This is attack technique is known as *code injection*. The attacker may also construct a sequence of return values by code injection, each pointing to an instruction already in memory, to result in an arbitrary chain of function calls, in what is known as a *code reuse attack*. Well-known code-reuse attacks include return-to-libc attacks [18] and Return-Oriented Programming (ROP) [19].

3) RETURN-ORIENTED ATTACKS

The first class of return-oriented attack is the return-to-libc attack, initially contributed by Alexander Peslyak in 1997 [20]. The return-to-libc attack consists in replacing the return address on the current call stack with the address of a function that is found within the executable memory of the process (e.g., within the `libc` C standard library), such as by properly exploiting a buffer overflow.

Later, more complex return-oriented attacks were developed. Generally, when exploiting return-oriented strategies, an attacker hijacks the control flow by exploiting a vulnerability such as a buffer overflow. Instead of injecting malicious code directly into the call stack, which may be detected and prevented from executing, ROP utilizes existing code snippets that are present in the benign program's memory, called *gadgets*, in order to perform the desired operations and reproduce arbitrary program behavior. Specifically, each gadget is a sequence of instructions ending with a `return` instruction. By chaining these gadgets together in the call stack, the attacker can redirect the program's execution to perform actions like modifying memory, executing system calls, or bypassing security mechanisms.

4) SIDE-CHANNEL ATTACKS

A side-channel attack is a security vulnerability that exploits unintended information leakage from auxiliary channels, allowing an attacker to infer sensitive data or access cryptographic keys. One aspect of side-channel attacks involves shared cache, a component in modern processors that stores data accessed by multiple cores.

In a shared cache side-channel attack, an attacker utilizes the behavior of the shared cache to extract useful information. When multiple cores access the cache simultaneously, their interactions can create observable patterns in their access times or state transitions. By carefully monitoring these side-channel effects, an attacker can deduce information about the data being processed by other cores [21].

One common shared cache side-channel attack is known as a cache timing attack. In this scenario, an attacker measures the time it takes to access specific cache lines, which can vary depending on whether the data is already present in the cache or needs to be fetched from the main memory. By repeatedly accessing certain cache lines and observing the access times, an attacker can determine patterns that reveal sensitive information, such as cryptographic keys.

Mitigating shared cache side-channel attacks often involves implementing countermeasures during system

design and software development. Techniques like cache partitioning, which segregates cache resources into different security domains, can help preventing leakage between processes or threads. Additionally, ensuring constant-time implementations and avoiding data-dependent control flows can reduce cache-related side-channel vulnerabilities.

Protecting against shared cache side-channel attacks requires a multi-faceted approach that combines hardware design, software development practices, and security-conscious programming techniques. To this end, this survey considers hardware facilities and software techniques designed to deal with memory integrity. Isolation or segregation techniques were analyzed by Gracioli et al. [22].

5) BRANCH TARGET INJECTION

The branch target injection exploit targets a processor's indirect branch predictor. Direct branches occur when the destination of the branch is known from the instruction alone at compile time. Indirect branches, on the other hand, occur when the destination of the branch is not known a priori, such as when the destination is read from a register or a memory location. The indirect branch predictor uses information about previously executed branches to predict the destinations of future indirect branches.

The utilization of indirect function calls is necessary when employing function pointers in compiled languages such as C and C++. For instance, function pointers are often used as an argument to sorting functions, in order to select a suitable comparison function, as shown in the following example. In the example, each call to `compare()` occurring within the `sort()` function will likely result in an indirect function call.

```
int compare(int a, int b)
{
    return a < b;
}
sort(array, &compare);
```

In addition to indirect branches explicitly performed by programmers, the compiler sometimes incorporates additional indirect branches without the programmer's explicit instruction. For example, in C++, calls to object functions often incorporate indirect calls, particularly when inheritance is applied. The following is an example of a scenario where the compiler might insert an indirect call, even without the presence of function pointers.

```
Vehicle *car = new Car();
car->drive();
```

The branch target injection exploit relies on influencing the speculated targets of indirect branches which allows the processor to execute instructions ahead of time. Under specific circumstances, attackers can manipulate the prediction mechanisms to redirect speculative execution to unintended target addresses.

Indirect `JMP` and `CALL` instructions consult the indirect branch predictor to direct speculative execution to the most likely target of the branch. By influencing these mechanisms, the attacker misleads the processor into speculatively executing instructions from the malicious code in a location accessible to the target process.

Indeed, the indirect branch predictor is a hardware structure, mostly transparent to the operating system, which is used to predict the destination of indirect branches ahead of actual instruction execution. The execution of malicious code through speculative means might allow accessing sensitive data or performing unauthorized operations. Although the outcomes of speculative execution are eventually disregarded if the speculation was incorrect, the processor's cache and other side channels could still expose sensitive information to the attacker [21], [23].

III. MEMORY SAFETY TECHNIQUES

A. RESEARCH SCOPE

In general, the purpose of a technique for memory safety is to prevent the attacker from writing to or reading from a protected memory area by exploiting vulnerabilities in the software or the hardware. Despite the amount of research in developing countermeasures, only a few protection techniques actually end up being implemented at the production level. Indeed, many of them remain research prototypes, often due to excessive overhead or complexity.

Requirements. Due to the variety of solutions proposed in the literature, this work focuses on techniques that satisfy the following requirements:

- The proposed technique has been adopted in production by one or more open-source operating systems, whether they are general-purpose operating systems (GPOS) or real-time operating systems (RTOS);
- At the production level, one or more toolchains (such as GCC and Clang) for C-family languages (C, C++, Objective-C, etc.) have provided support for the proposed technique
- The technique, if hardware-assisted, is fully implemented at the processor level, without requiring any specific external mechanism such as a Trusted Platform Module (TPM) or other specialized modules that are only available on certain commercial platforms.

Threat models. We have listed a set of assumptions that we used to restrict the scope of our research, setting the bounds of the possible threat model with the assumed capabilities of the attacker. The survey does not cover protection techniques against other types of threats that require different capabilities.

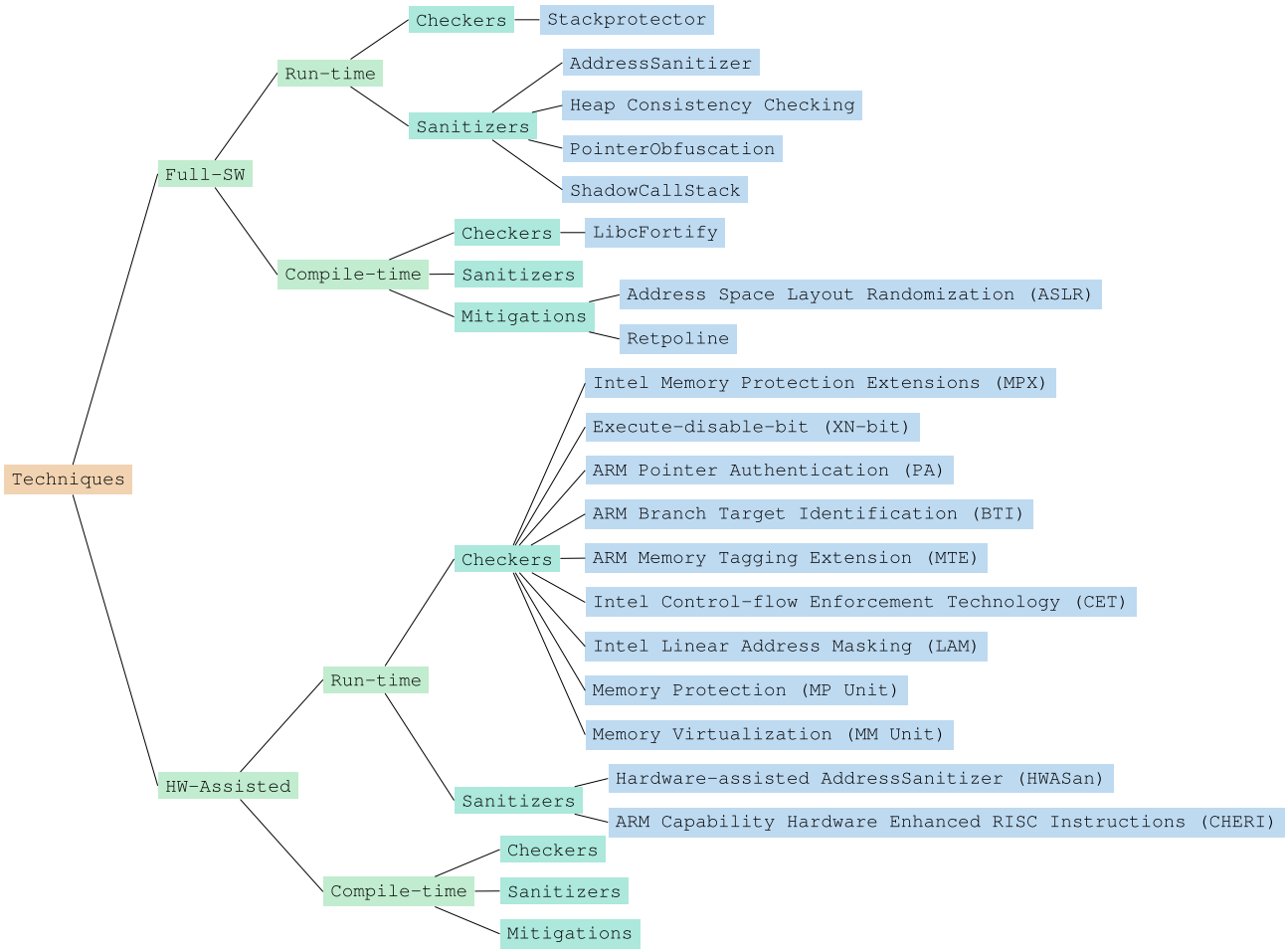

FIGURE 2. Taxonomy of the considered protection mechanisms.

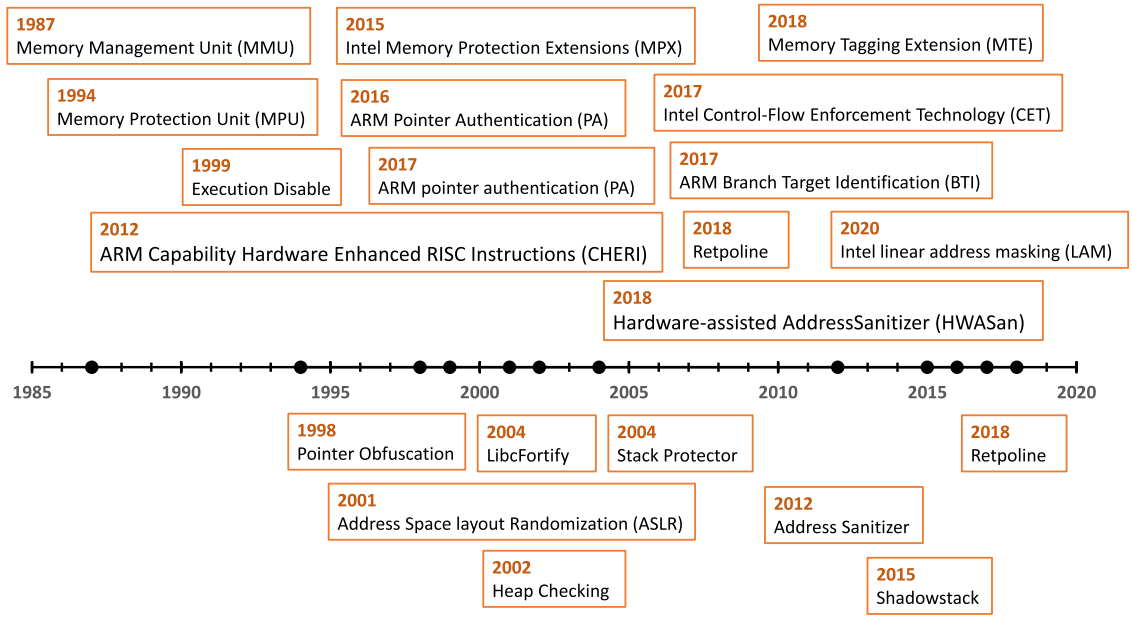

FIGURE 3. Timeline of protection techniques.

Software threat model:

- The user space memory content and layout are readable by the attacker;
- Memory errors (such as buffer overflows or dangling pointers) can be present inside the program due to bad programming practices or vulnerabilities in imported dependencies;
- The attacker can write non-code segments by exploiting a memory error or other vulnerabilities.

Hardware threat model:

- The hardware is trusted (e.g., the hardware has not been substituted with faulty or modified hardware);
- The attacker cannot obtain physical access to the device;
- Hardware glitches cannot occur (i.e., unexpected faulty behaviors occurring spontaneously or by tampering with the hardware and leading to temporary vulnerability to certain attack classes).

Taxonomy and timeline. Figure 2 provides a visual taxonomy of the techniques investigated by this survey. Figure 3 provides a timeline starting from 1985 representing the years in which the most relevant countermeasures discussed in this survey were first developed.

In the following, relevant techniques are presented, starting from techniques implemented fully in software before moving to hardware-assisted countermeasures requiring specific functionality at the processor level.

B. FULL SOFTWARE PROTECTIONS

1) RUNTIME CHECKERS

StackProtector: The first proposal of a mechanism to prevent stack overflow attacks dates back to 1998, when Cowan et al. [24] presented *StackGuard*, which was then released as a set of patches for the GCC compiler toolchain. The main idea of StackGuard is to place a randomly generated integer, called *stack canary*, between any stack-allocated buffers and the return address saved on the stack. Then, before a routine uses the return pointer on the stack, the value of the canary is checked to make sure that it has not been changed This makes it more difficult to correctly execute a stack overflow attack, because overwriting the return pointer by exploiting a stack overflow vulnerability would also require overwriting the value of the stack canary. The terminology is due to an analogy with coal mine canaries, given that stack canaries are used to determine whether it is safe to carry on the execution of the program.

At the beginning of the 2000s, Etoh [25] from IBM implemented *ProPolice*, improving the idea of StackGuard by placing buffers after local pointers and function arguments in the stack frame. In 2005, Henderson suggested a less intrusive implementation of the mechanism [26], which has been included as a compile option in GCC starting with version 4.1 [27].

This implementation has been optimized over the years and is now tunable to enable management of performance tradeoffs [28], but still retains the same overall working principle. Indeed, the current version of the GCC stack protector works by inserting stack canaries on the stack frame of certain functions and is still used today due to its simplicity and low runtime overhead. In the current implementation, as illustrated in Figure 4, the canary is placed right after local variables, protecting both the old frame base pointer and the return addresses from direct overflows. Furthermore, the mechanism arranges the local stack variables to ensure `char` buffers are always allocated next to the canary. This assumption prevents a direct overflow from corrupting other local variables.

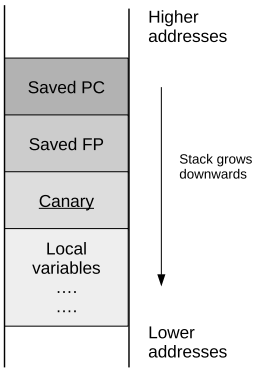

FIGURE 4. Conventional stack frame layout when stack protector is enabled.

Dang et al. [29] reported the results of an experimental evaluation of the overheads incurred when using the GCC stack protector, and found that when protecting all functions it can reach a cost of up to 10% overhead, expressed in terms of additional CPU time.

2) RUNTIME SANITIZERS

AddressSanitizer: The *AddressSanitizer* (also known as ASan) is an open-source memory error detector originally introduced by Serebryany et al. from Google [30]. ASan works as a compiler instrumentation module and is currently implemented in Clang (starting from version 3.1 [31]) and GCC (starting from version 4.8 [32]). ASan targets the most common instruction set architectures, including x86 and ARM, both in their 32-bit and 64-bit variants. The tool consists of an additional compiler pass and a related runtime library. It was designed to find and catch memory errors such as use after free, stack and heap overflows, use-after-return or use-after-scope vulnerabilities.

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, 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 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 memory classes have a correspondence built so that the

shadow memory mapping (called `mem_to_shadow`) can be computed efficiently. 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 red zones; hence, the wider the red zone, 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 will result in the following:

```
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.

Heap Consistency Checking: Heap Consistency Checking [33] is an extension of the `libc` library to help debug and detect memory-related errors in C programs by performing consistency checks on allocated dynamic memory blocks. These consistency checks, implemented in the `libc_malloc_debug` library, aim at detecting errors related to memory management. For example, they can identify if a memory block is freed more than once or if the bookkeeping data structures preceding an allocated memory block are corrupted. To enable the consistency check, the programmer must call the `mcheck()` function before performing a memory allocation with `malloc()`, and preload the malloc debug library. The `mcheck()` function installs debugging hooks for the memory-allocation functions. These hooks enable occasional consistency checks on the state of the heap. Linking a program with the `-mcheck` flag inserts an implicit call to `mcheck()` (with a `NULL` argument) before the first memory allocation function call. The `mcheck_pedantic()` function is similar to `mcheck()`, but it performs checks on all allocated blocks whenever any memory allocation function is called. However, this thorough checking can significantly slow down the program's execution. If the system detects an inconsistency in a heap while performing the checks, a user-defined function provided is invoked. as in the following example:

```
mcheck();
p = malloc(N);

free(p);
free(p);  // Aborted (block freed twice)
```

The usage of additional consistency checks, such as those provided by `libc_malloc_debug`, can help prevent the exploitation of numerous vulnerabilities, especially when used in conjunction with a memory allocator debugging tool [34].

Pointer Obfuscation: Pointer obfuscation encompasses a range of security techniques employed in common programming languages that support explicit usage of pointers. These methods play a crucial role in complicating the task of discerning which function is invoked when a function pointer is utilized. Their primary objective is to obscure both pointers and memory addresses, rendering it challenging for potential adversaries to anticipate or manipulate these critical elements.

One such technique is pointer encryption, where pointers are encrypted before being stored in memory and decrypted only when loaded into CPU registers. This method significantly complicates an attacker's ability to interpret or manipulate pointer values [35].

Another approach to obfuscate function pointers is the creation of a globally accessible array to store pointers for each function. This technique replaces conventional function calls by utilizing array indexing to access the relevant function pointer, ensuring the execution of the desired function [36].

These obfuscation techniques can be fortified with supplementary methods, including value encoding/aliasing and data structure/code obfuscation, to further mask the underlying structure of the function pointer [37], [38], [39], [40].

glibc fortification: The `FORTIFY_SOURCE` macro, offered by the GNU C library (*glibc*), provides a lightweight form of protection against buffer overflows by adding an extra validation layer to *glibc* functions that operate on memory and strings. The macro can be set to three different levels. When the macro is enabled, the number of bytes that will be copied from a source to a destination during certain operations are computed in advance and then checked during the memory copy operation. For instance, when using `strcpy()` to copy the contents of a string to another memory location, the macro calculates the size of the data being copied. If an attacker tries to copy more bytes than can fit into the destination buffer, the macro detects this attempt and halts

the execution of the program. The macro was first introduced in 2004 by Red Hat engineers as a set of GCC patches [41]. Then, after being merged in the GCC mainline, it is still developed and maintained by Red Hat in GCC/Clang on Linux [42]. The implementation of a function fortified with FORTIFY_SOURCE is conceptually similar to a wrapper function like the following:

```
strncpy(d, s, n)
{
    d_sz = __builtin_object_size(d);

    if (d_sz == ESTIMATE_FAIL)
        // built-in fallback policy
        return;

    if (d_sz < n)
        __chk_fail();
    else
        __original_strncpy(d, s, n);
}
```

In the example, the strncpy() wrapper accepts three parameters: a pointer to the destination memory buffer (d), a pointer to the source memory buffer (s), and the number of bytes to be copied (n). Then, if the size of the destination buffer is smaller than the number of bytes to be copied from the source buffer, then the copy operation is aborted. Otherwise, the original version of strncpy(), __original_strncpy(), is invoked.

The core functionality is based around the object size built-in function (__builtin_object_size), which returns a constant estimate for the size of an object, computed at compile time. A dynamic built-in function (dynamic_object_size) is provided in the LLVM compiler toolchain, starting with version 9 [43]. This function is used in substitution to the constant object size function when the protection level of FORTIFY_SOURCE is set to 3, which is the maximum level. FORTIFY_SOURCE levels 1 and 2 rely on constant object sizes; therefore, the runtime overhead is negligible. On the other hand, FORTIFY_SOURCE with protection level 3 uses dynamic expression computation. Given that computing the object size can become arbitrarily complex depending on the data structures selected by the programmer, the runtime overhead can also increase significantly with an increase in object complexity.

An alternative to FORTIFY_SOURCE, named OpenOSC (which stands for Open Object Size Checking), was proposed by Cisco engineers [44]. OpenOSC and FORTIFY_SOURCE are both built upon the compiler's built-in function to determine object size, thus provide equivalent memory overflow detection and protection capabilities. Although the two protection mechanisms can coexist in a Linux-based system, each specific package can only be compiled with either OpenOSC or FORTIFY_SOURCE protection, but not both.

ShadowCallStack: ShadowCallStack is a compiler instrumentation pass designed primarily for the ARM 64-bit architecture (*aarch64*), aimed at safeguarding programs against return address overwrites, typical of stack buffer overflows. ShadowCallStack is available for both GCC and Clang. When ShadowCallStack is enabled, the function's return address is stored into a separately allocated shadow call stack during the function prologue of each non-leaf function. Then, the return address is retrieved from the shadow call stack during the function epilogue. The return address is also stored on the regular stack to ensure compatibility with stack unwinding, but remains otherwise unused by the function call and return mechanisms.

The aarch64 implementation of ShadowCallStack is considered ready for production use and has been integrated into the Android libc runtime. An implementation of ShadowCallStack for the x86_64 architecture was initially provided in LLVM but exhibited critical performance and security shortcomings, leading to its removal from LLVM starting with version 9 [45].

3) COMPILE TIME MITIGATIONS

Retpoline: Return trampoline (Retpoline) is a protection measure against branch target injection exploits, primarily targeting indirect branches [46]. These indirect branches are predicted using information from prior branch executions, and the attack in question involves tampering with the execution of such indirect branches, like the JMP instruction, to extract sensitive data that resides outside the user's authorized permissions. This sensitive data could include confidential cryptographic keys, and the attack is achieved by influencing the anticipated destinations of these indirect branches [46].

Consider the following example, where a jump is executed to an instruction address stored in the %rax register. The retpoline sequence functions through multiple stages to disentangle speculative execution from non-speculative execution.

```
    // ...
    // indirect call instruction
    call escape_speculation
.non_vulnerable_sequences:
    pause ; LFENCE
    jmp non_vulnerable_sequences
.escape_speculation:
    mov %rax, (%rsp)
    ret
```

The call escape_speculation instruction pushes the address of non_vulnerable_sequences onto the stack and the Return Stack Buffer (RSB). Then, the mov instruction writes over the return address stored on the stack.

At this point, there is a divergence between the in-memory stack and the RSB. If the processor speculates, it utilizes the RSB entry it created and jumps at the `pause` instruction, where it becomes *trapped* in an infinite loop. Eventually, the processor realizes that the speculative return does not match the in-memory stack value, leading to the halt of speculative execution [47]. The performance assessment of retpoline indicates that the instructions have a negligible impact on overall performance, a factor heavily contingent on the specific implementation strategy of a given architecture [47], [48].

ASLR: Address Space Layout Randomization (ASLR) is defense technique which is based on randomizing the position of key data areas within the address space of a process. By doing so, ASLR makes it difficult for an attacker to predict the location of specific memory regions, such as functions to be exploited, making it more difficult for them to carry out successful attacks. ASLR works by rearranging the positions of important data areas in the address space of a process. These areas typically include the base address of the executable, the stack, the heap, and libraries. By randomizing their positions, ASLR ensures that these key components are located at different memory addresses each time the process is run. The concept of ASLR was first introduced by the Linux PaX project, which coined the term and released the initial design and implementation of ASLR as a patch for the Linux kernel in July 2001 [49]. The main goal of ASLR is to increase the security of a system by significantly reducing the likelihood of successful attacks. Since an attacker needs to know the exact memory addresses of specific components to exploit vulnerabilities, the randomization introduced by ASLR makes it highly improbable for them to guess the correct locations. By increasing the search space and making it harder to predict memory layouts, ASLR adds an extra layer of defense against memory-based attacks.

ASLR is highly effective against many types of attack; however, since the majority of modern processors have at least one shared level of cache, several research papers showed that ASLR can be bypassed on modern cache-based architecture [50], [51], [52]. As a result, an attacker can derandomize virtual addresses of a victim's code and data by locating the cache lines that store the page-table entries used for address translation.

C. HARDWARE-ASSISTED

The relevance of security in modern operating systems pushed chip designers to introduce several security-related hardware facilities in their processors. Historically, hardware vendors introduced a general memory protection mechanism (e.g., MMU, MPU). In recent years, on the other hand, hardware manufacturers have tried to introduce transparent and lightweight mechanisms to enforce memory protection and control-flow integrity. Some relevant examples are ARM Pointer Authentication or Intel Control-flow Enforcement, detailed in the following section.

1) RUNTIME CHECKERS

ARM Pointer Authentication: ARM Pointer Authentication (PA) is a hardware feature that is included in version 8.3 of the ARMv8 processor architecture [53]. In a nutshell, ARM 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.

In 64-bit architectures, not all 64 bits of a pointer are used to address memory locations. Typically, a smaller number of bits, such as 48, are sufficient to address the entire memory space. This means that a significant portion of the 64-bit pointers remains unused. For example, on an ARMv8-A Linux kernel (aarch64), only the least significant 48 bits of a pointer are used for addressing. ARM's PA implementation takes advantage of the unused most significant bits of memory addresses. It uses a portion of these bits, specifically 16 bits, to store a Pointer Authentication Code (PAC). PA embeds an authentication code, the PAC, within the authenticated pointer itself.

ARM PA leverages the Qualcomm ARM Authenticator (QARMA), which is a specialized lightweight tweakable block cipher. Specifically, the PAC is a cryptographic checksum obtained by truncating the QARMA algorithm's output. QARMA ensures authenticity and integrity through tweaks, where the permutation computed by the algorithm on the plaintext depends on a secret key and an additional salt value. The PAC is computed using three inputs: **(i)** the memory pointer value to be authenticated, **(ii)** a secret key stored in dedicated processor registers, and **(iii)** context information that specifies where the authenticated pointer can be used. An example of context information is the stack pointer, which associates the authenticated pointer with the stack frame of a specific function.

Pointers that include both the memory pointer and the PAC are referred to as signed pointers. Before using signed pointers, they need to be authenticated. The authentication process involves recomputing the PAC for the pointer using the key and context information and comparing it with the PAC stored in the signed pointer. This ensures the integrity and validity of the pointer.

To support the creation and authentication of PACs, ARMv8.3-A introduced a set of processor registers to store the required keys. The specification defines five 128-bit registers for this purpose. The keys are categorized as either type A or type B, and their specific semantics are left to the programmer to define. The ARMv8.3-A architecture extends the instruction set architecture (ISA) with additional

instructions to facilitate the creation and authentication of PACs. Two sets of instructions, PAC* and AUT*, are introduced for this purpose. The PAC* instructions are used for creating PACs, while the AUT* instructions are used for authenticating PACs. Linux and GCC/LLVM support ARM Pointer Authentication. An attempt to extend the support was proposed by various research groups [54], [55]. This mechanism is known to be weak against timing attacks [56]. Additionaly, an attempt to emulate the mechanism to support older generations of processors was proposed and implemented for FPGA-enabled SoCs [57].

ARM Branch Target Identification: ARM also introduced a Branch Target Identification (BTI) extension to their architecture, which is accessed with the BTI instruction [58]. This instruction must be used to mark valid targets of indirect branches. When the memory page is set as guarded, the processor traps to an exception if a control-flow instruction performs an indirect branch to any instruction other than those marked as valid with BTI. The BTI mechanism can secure indirect branches, enforcing that the destination location of the branch contains only instructions from an acceptable list. Combining the PA and BTI countermeasures allows for solid forward-backward control-flow integrity, reducing the possibility of an attacker hijacking the execution flow to execute arbitrary code. Nonetheless, the `RET` instruction is not considered an indirect branch control-flow instruction; thus, the BTI mechanism does not protect it. Furthermore, direct branches and calls are also left unprotected as the code segment is assumed to be non-writable.

Execution Disable XN: Since version 6 of their architecture, ARM introduced the XN (eXecute-Never) [59] bit feature, which prevents the 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. A similar feature is provided by other chip designers, including Intel [60], which offers the Execute Disable Bit (XD-bit), while AMD calls it the NX-bit in their implementations.

ARM MTE: ARM Memory Tagging Extension (MTE) is a security feature introduced in ARMv8.5, providing a hardware memory tagging (memory coloring) mechanism. MTE provides enhanced protection against memory vulnerabilities (e.g., buffer overflow) by associating tags for each memory operation (e.g., pointer referencing a memory location). On each load/store operation, the associated tags for the memory and the referenced address are validated for uniformity, and, upon mismatch, a memory violation is detected.

The implementation of MTE is only available on 64-bit systems, since it requires the Top-Byte-Ignore feature, which is only available on 64-bit architectures and results in the hardware ignoring the top byte of a pointer when accessing memory. The MTE design divides physical memory into 16-byte granules with 4-bit associated tags. These 4-bit tag values are associated with any memory operation targeting that specific memory area. MTE can be used to detect/mitigate memory errors like buffer overflows and use-after-free in memory-unsafe languages such as C and C++. To take advantage of MTE, software stacks must be updated to guarantee proper functionality and interoperability. For example, in Glibc, functions such as malloc, free, realloc are rewritten to interact with tag values, whereas operating system kernels such as Linux provide explicit support for MTE capabilities [61], [62].

As per specification, MTE introduces a set of new processor registers to configure and retrieve information on the tagging mechanism. MTE can offer two modes of operation: synchronous and asynchronous. Following any tag mismatch, a hardware exception is raised in the synchronous mode, allowing precisely determining the faulty instruction. This mode provides better granularity in error detection at the cost of introducing a higher performance overhead. On the contrary, when the configuration is set to be asynchronous, certain information is accumulated in system registers (TFSR_ELx), enabling the system to react to tag mismatches in an asynchronous manner (e.g., in context switch). This allows reducing the overhead at a cost of a loss in accuracy, since the violations are isolated to a particular thread of execution [63], [64]. Nonetheless, in terms of performance, some CPUs demonstrate comparable MTE performance between stricter and less strict tag checking modes, making a small performance slowdown acceptable.

Intel MPX: Intel MPX (Memory Protection Extensions) [65] was a set of extensions introduced by Intel for the x86 instruction set architecture. It aimed to enhance software security by providing runtime checks on pointer references that could be maliciously exploited due to buffer overflows. Intel MPX, with compiler [66], runtime library, and operating system support [67], attempted to enhance the security of software by checking pointer references whose normal compile-time intentions are maliciously exploited at runtime due to buffer overflows by detecting and preventing runtime memory corruption vulnerabilities by validating pointer accesses.

The extensions introduced new bounds registers and instruction set extensions that operated on these registers. Additionally, there were *bound tables* to store bounds beyond what could fit in the bounds registers. Intel MPX utilized four new 128-bit bounds registers. Each of these registers stored 64-bit values representing the lower and upper bound for the memory areas of a buffer. Specific instructions were provided to store (`bndmk`) and check (`bndcl`, `bndcu`) values from those registers. The architecture also included configuration registers and a status register. These registers facilitated the management of bounds, provided information about memory addresses, and reported error codes in case of exceptions.

Intel MPX utilized a two-level address translation approach to store bounds in memory. The top layer was a Bounds Directory (BD) created during application startup. Each BD entry was either empty or contained a pointer to a dynamically created Bounds Table (BT). The BT contained a set of pointer bounds and the linear addresses of the pointers. Special instructions like `bndldx` (bounds load) and `bndstx` (bounds store) transparently performed the address translation and accessed bounds in the appropriate BT entry.

An example application of Intel MPX to protect access to a memory buffer is given in the following piece of code.

```
// Original code
type_t  t[10]
type_t* pt = t;
type_t  val;

for (i = 0; i < N; i++) {
    pt  = pt + i;
    val = *pt;
}
// MPX enabled
type_t  t[10]
t_b = bndmk t, t+79 // bound in bytes
type_t* pt = t;
bndcl t_b, pt       // load lower bound
bndcu t_b, pt+7     // load upper bound
type_t  val;


for (i = 0; i < N; i++) {
    pt  = pt + i;
    bndcl t_b, pt
    bndcu t_b, pt+7
    val  = *pt;
    val_b = bndldx pt; // bound check
}
```

However, despite its initial promise, Intel MPX faced several flaws and limitations in its design. As a result, support for Intel MPX has been deprecated or removed from most compilers and operating systems. Intel officially listed MPX as removed in 2019 [68]. Oleksenko et al. [69] performed a detailed root cause analysis of issues in the Intel MPX architecture design through a cross-layer dissection involving the hardware, operating system, compilers, and applications, showing program slowdown up to 2x when using Intel MPX.

Intel CET: Control-Flow Enforcement Technology (CET) is a security feature developed by Intel [70] to enhance protection against exploits that target the control flow of running programs, such as Return-Oriented Programming (ROP) and Jump-Oriented Programming (JOP). CET introduces two primary security mechanisms that can be individually enabled for different privilege levels [71], [72]. CET includes the following features:

- Indirect Branch Tracking (IBT): IBT is a feature designed to mitigate control-flow hijacking attacks. It enforces that indirect branches within the program (branches without fixed targets at compile time) can only jump to valid and authorized destinations. This prevents attackers from diverting the control flow to arbitrary code locations. IBT introduces new instructions (`ENDBR32` and `ENDBR64`) to be used by the compiler to mark legitimate targets for indirect jumps. These instructions are decoded as `NOP` on legacy processors to preserve backward compatibility. Indirect jumps not marked by `ENDBR32/64` will trigger an exception. Separate state machines are used to track indirect calls for both user and supervisor modes, and they can be set to `'no_track'` mode to reduce system resource utilization. Compilers also offer the option to limit the usage of `ENDBR` instructions to reduce code size.
- Shadow Stack (SHSTK): Shadow stack is a hardware-managed stack created by the operating system alongside the standard call stack, which is unique for each privilege level. During execution, each `CALL` instruction pushes the return address onto both the normal and the shadow stacks, and the `RET` instruction then pops these return values from both stacks. This configuration enables the generation of Controlflow Protection (CP) exceptions in cases of conflicting values, providing enhanced protection against control-flow attacks.

One notable advantage of CET lies in its seamless integration, requiring no code instrumentation by programmers. Thanks to its negligible overhead, this method does not have a significant impact on application performance [73]. Moreover, the incorporation of CET functionality into the Linux kernel and mainstream compilers demonstrates its increasing adoption and integration in development workflows.

Intel LAM: The Intel Linear Address Masking (LAM) feature [74] introduces a modification to the validation process for 64-bit linear addresses, enabling software to utilize the unaltered address bits for metadata purposes. In the context of 64-bit mode, where linear addresses consist of 64 bits, they undergo translation using either 4-level paging (which translates the lower 48 bits) or 5-level paging (translating 57 bits). The upper bits of linear addresses are set aside for *canonicality*. A linear address is considered 48-bit canonical when bits 63 to 47 of the address are the same, whereas it is 57-bit canonical when bits 63 to 56 are identical. Importantly, any linear address that is 48-bit canonical is automatically 57-bit canonical. In scenarios where 4-level paging is active, the processor mandates that all linear addresses used for memory access

must be 48-bit canonical. Similarly, 5-level paging ensures that all linear addresses are 57-bit canonical. However, for software applications that link metadata to a pointer, there is a potential advantage in positioning metadata in the upper (untouched) bits of the pointer itself. The challenge arises from the requirement of canonicality enforcement, which implies that software would need to mask the metadata bits in a pointer to make it canonical before using it as a linear address for memory access. LAM addresses this issue by allowing software to employ pointers with metadata without requiring manual masking of the metadata bits. With LAM enabled, the processor automatically masks the metadata bits in a pointer before using it as a linear address for memory access.

ARM Cheri: The Capability Hardware Enhanced RISC Instructions (CHERI) feature extends the ARM architecture with the primary goal of enhancing security and memory protection to address widely exploited vulnerabilities. CHERI diverges from conventional approaches by introducing capabilities that are unforgeable and constrained references for memory access. These capabilities are associated with every data object in memory, encompassing information about the object's type, permissions, validity status, and bounds (defining the address space within which the capability authorizes loads, stores, and/or instruction fetches). Additionally, CHERI introduces capability-aware instructions that facilitate interactions with capabilities attached to each object [75], [76].

It is worth highlighting that CHERI is still in the research phase and has not yet seen full integration into commercial systems. Nonetheless, it is designed to coexist with established security practices and can serve as an additional protective layer alongside traditional methods like segmentation and paging. CHERI introduces two distinct operating modes: **(i)** the hybrid mode [77] and **(ii)** the pure-capability mode [78]. In the hybrid mode, only part of the system code is converted to using capabilities. In the pure-capability mode, the system exclusively relies on capabilities for managing memory access, eliminating the need for conventional memory protection mechanisms. However, the transition to a pure-capability model may entail software modifications, including adjustments to the operating system and application code to benefit from this security model fully. The CHERI capability model implementation complexity is sufficiently low for consideration on modern processors, and performance is noticeably faster than weaker enforcement in software.

Moreover, the introduction of broader data pipelines has an impact on reducing the system's clock speed and degrading the overall computational performance. Woodruff et al. [78] compared the overhead introduced by the CHERI model against different memory protection hardware techniques, such as Intel MPX, in terms of memory footprint and the number of memory references.

HWASan: Hardware-assisted AddressSanitizer (HWASan) functions as a memory error detection tool similar to AddressSanitizer. It uses the hardware memory tagging feature to reduce the memory consumption footprint of AddressSanitizer (AS) significantly. The fundamental idea behind HWASan bears similarities to the coloring mechanism introduced in memory tagging (MT) [61], wherein both pointers and memory addresses are associated with a random tag value. The validity of a memory access in HWASan hinges on the matching of these tags between pointers and memory. Its primary aim is to ensure the integrity of data and memory, such as keeping addresses within predefined bounds and preventing references of pointers after they have been freed. To accomplish this, HWASan leverages the ARM architecture's `Top-Byte-Ignore` [79] feature, which enables the insertion of compiler-time checks for `load` and `store` operations. As an example, the tagging of heap memory and pointers involves the use of a modified version of the `malloc` function to instrument memory allocation and monitor how that memory is utilized.

Consequently, proper compiler support is essential to insert the requisite instructions, enabling memory checking, and delivering detailed information about memory errors [80], [81]. HWASan demonstrates strong compatibility with existing code bases, particularly when compared to pure hardware solutions. Nonetheless, it is important to highlight that HWASan does introduce some CPU overhead, approximately doubling the overhead in comparison to conventional operations, along with a moderate increase in code size and memory footprint (RAM) [82], [83]. An avenue for improvement lies in optimizing compilers further to avoid unnecessary tagging of variables that are not affected by vulnerabilities.

IV. EVALUATION

Over the years, researchers have designed several kinds of protection techniques. Consequently, together with vulnerability databases, security testbeds were also developed to measure the effectiveness of those defence techniques. While most of defense methods can be evaluated from the point of view of runtime overhead, no standard benchmark allows assessing the effectiveness and robustness of a countermeasure. This section presents an overview of existing techniques utilized for evaluating memory integrity techniques and a qualitative comparison among the techniques surveyed in Section III.

A. DATABASES OF FLAWED SOFTWARE

Several databases encompassing flawed software were created over the years to test the effectiveness of protection techniques. The Software Assurance Reference Dataset (SARD) [84] is a growing database maintained by the National Institute of Standards and Technology (NIST),

consisting of approximately 170,000 test programs with a set of known security flaws. These test cases include designs, source code, and binaries from all the phases of the software life cycle, mainly written in C, C++, Java, PHP, and C# and covering 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 suites via the SARD online interface or search for test cases by vulnerability kind, name, size, keywords, and other parameters. Many tests include non-vulnerable or benign program code to test for false positives, in which flaws are resolved in advance. Each test case is described in SARD using metadata, encompassing 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, meaning 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 can be added to supersede it. Among the SARD database, the Juliet test suite [85], targeting the C and C++ languages, is one of the most relevant and comprehensive datasets.

Another popular database was created by the Intelligence Advanced Research Projects Activity (IARPA) in the context of the Securely Taking On New Executable Software of Uncertain Provenance (STONESOUP) project [86]. For that project, MITRE created a test infrastructure and a test suite consisting of test programs purposely containing memory errors. In the database (which is currently available on the SARD webpage, under the STONESOUP name), multiple test cases are available for each class of vulnerability, and relevant sets of benign and malicious inputs are provided for each test case. This database was utilized to evaluate numerous tools, including advanced prototypes developed by Symantec [87]. While both the Juliet Test Suite and IARPA STONESOUP Test Suite involve similar classes of vulnerabilities, the Juliet Test Suite comprehends a broader range of cases, while the latter is more tailored to evaluating the security of software through dynamic analysis, especially when dealing with data from untrusted sources (i.e. network, file, etc.).

B. SYNTHETIC BENCHMARK SUITES

When assessing the resistance of defense mechanisms against vulnerabilities, the absence of standard benchmarks poses a challenge. A database of flawed software and vulnerabilities, such as those mentioned above, can be used to construct customized testbeds to evaluate resistance against a specific flaw or vulnerability. However, this approach is lacking in terms of generality and automation, as it heavily relies on the specific subset of tests chosen for the evaluation. Therefore, besides test databases, other research efforts have been spent into trying to standardize test benchmarks for general-purpose computing and emerging embedded platforms through synthetic benchmarking. Synthetic benchmarks, generally, are benchmarking tools created artificially and run in a monitored environment, to assess the performance or capabilities of a solution. They offer a straightforward solution and an easy setup, providing a standardized basis for comparing different approaches. This not only simplifies the evaluation process but also ensures consistent and comparable assessments across different scenarios. The leading example is represented by the *Runtime Intrusion Prevention Evaluator* (RIPE), developed in 2011 by Wilander et al. [88] as an extension of a previous prototype released in 2003. RIPE is a synthetic testbed suite comprising more than 800 buffer overflow patterns. The purpose of RIPE is to evaluate the coverage of any given countermeasure by performing a range of buffer overflow attacks and recording their success or failure. RIPE has been used to demonstrate the effectiveness of several tools and security techniques [89], [90], [91]. RIPE was released under the MIT License to facilitate the comparison between different countermeasures; however, it only supports the `i386` processor architecture.

Since the release of RIPE, a consistent number of derived extensions and research works have been produced. RIPE-ARM [93], released in 2020, is an implementation of the RIPE benchmark targeting ARMv7 32-bit platforms. Unfortunately, RIPE-ARM was not publicly released. In 2022, Calatayud and Meany [94] worked on a comparative analysis of buffer overflow vulnerabilities in high-end IoT devices. The authors modified the original RIPE by replacing the architecture-specific shellcode for code injection attacks implemented in RIPE with shellcode compatible with architectures typical of the IoT domain, although still only targeting 32-bit operating systems. The resulting framework, however, is only compatible with a specific set of IoT platforms and with the FreeRTOS operating system. Wang et al. [95] presented a port of RIPE for platforms based on the RISC-V architecture, for the purpose of evaluating a defense technique presented in the same paper. Unfortunately, the related source code is not publicly available. Roascio et al. [96] presented Em-RIPE, a tool that extends the original RIPE approach to ARM-based microcontrollers. The code of Em-RIPE is publicly available[2] and is organized as a proof-of-concept for 32-bit ARM architectures, empowered by the FreeRTOS operating system. The most recent extension of the RIPE benchmark is X-RIPE [97]. X-RIPE is an overhaul of the original RIPE project to target multiple processor architectures, with the objective of providing a quantitative evaluation of the protection coverage offered by a specific mechanism against buffer overflows. X-RIPE supports `i386`, `x86-64` and `aarch64` architectures. These quantitative benchmarking tools, however, are synthesized testbeds, deliberately vulnerable with the sole purpose of conducting attacks against themselves. Therefore, they

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

TABLE 1. Comparison of the presented techniques. The comparison is both qualitative and quantitative (whenever possible).

Name	Reference	Year	Type	Architecture	Required support	Category	Effective against	Overhead
StackProtector	[26]	2004	Full-software	i386, x86, arm, aarch64, riscv	Compiler	Run-time Checker	Flow corruption (Stack)	Non-negligible [29]
Address Sanitizer	[30]	2012	Full-software	x86, aarch64	Compiler & Library	Run-time Sanitizer	Flow/data corruption	Non-negligible [69]
Heap Consistency Checking	[33]	2002	Full-software	i386, x86, arm, aarch64, riscv	Compiler	Run-time Sanitizer	Flow/data corruption	-
PointerObfuscation	[38]	2003	Full-software	x86	Compiler	Run-time Sanitizer	Pointer corruption	-
glibc fortification	[41]	2004	Full-software	x86	Compiler	Run-time Sanitizer	Flow/data corruption	-
ShadowCallStack	[45]	2019	Full-software	x86, aarch64	Compiler	Run-time Sanitizer	Flow corruption (Stack)	Non-negligible [45]
Retpoline	[46]	2018	Full-software	x86, aarch64	Compiler	Compile-time Mitigation	Branch-target corruption	Negligible [47]
ASLR	[49]	2001	Full-software	x86, aarch64	Compiler & OS	Run-time Mitigation	Flow/data corruption	Negligible [92]
Intel MPX	[65]	2015	Hw-assisted	x86	Compiler & Library	Run-time Checker	Flow/data corruption	Non-negligible [69]
Intel Execute Disable Bit	[60]	1999	Hw-assisted	i386, x86	Compiler	Run-time Checker	Flow/data corruption	-
ARM Execution Disable XN	[59]	1999	Hw-assisted	arm, aarch64	Compiler	Run-time Checker	Flow/data corruption	-
ARM Pointer Authentication	[53]	2016	Hw-assisted	aarch64	Compiler & OS	Run-time Checker	Flow/data corruption	Negligible [54]
ARM Branch Target Identification	[58]	2018	Hw-assisted	aarch64	Compiler & OS	Run-time Checker	Flow/data corruption	-
ARM Memory Tagging	[63]	2020	Hw-assisted	aarch64	Compiler & OS	Run-time Checker	Flow/data corruption	-
Intel CET	[70]	2020	Hw-assisted	x86	Compiler & OS	Run-time Checker	Flow/data corruption	Negligible [73]
Intel LAM	[74]	2020	Hw-assisted	x86	Compiler & OS	Run-time Checker	Flow/data corruption	-
ARM Cheri	[76]	2019	Hw-assisted	aarch64	Compiler & OS	Run-time Sanitizer	Flow/data corruption	Negligible [78]
HWAsan	[82]	2019	Hw-assisted	aarch64	Compiler & OS	Run-time Sanitizer	Flow/data corruption	Non-negligible [82]

provide a valuable benchmark only when dealing with basic attack techniques. Compared to other approaches, they offer no evaluation of complexity or performance. Therefore, they can be used as automatic testing instruments only if coupled with qualitative and performance evaluations.

C. QUALITATIVE COMPARISONS

Several works tried to compare defense techniques from the qualitative point of view. Szekeres et al. [98] attempted to organize the knowledge about various protection techniques by setting up a general model for memory corruption attacks, comparing different methods to help designers of new protection mechanisms in finding the right balance between effectiveness and efficiency. Another notable example is the work by Kisore [99], which tried to formalize the general requirements that a protection technique shall implement, such as interoperability with legacy software, scalability, and low overhead. Then, the set of requirements was applied to evaluate well-known buffer overflow protection techniques from the qualitative point of view, providing an overall summary of the strengths and weaknesses of each technique.

D. OUR COMPARISON - HYBRID APPROACH

When considered individually, qualitative and quantitative approaches do not offer a complete way of evaluating a technique. In Table 1, we provide a summary and comparison of the techniques presented in this work, from both the quantitative and qualitative point of view. Note that it was not possible to gather all the information for some of the techniques due to missing hardware or lack of support by evaluation existing tools.

V. FUTURE PERSPECTIVES

Throughout the years, software engineers have devised numerous measures to address memory safety. The various techniques presented in this survey represent only a tiny part of the research and effort put in place by researchers and companies during these years. This section delves into pivotal insights and emerging trends regarding memory safety. From hardware advancements to the rise of memory-safe programming languages, evolving strategies for mitigating vulnerabilities are considered, and an overall outline is defined for addressing memory safety concerns.

A. CURRENT LIMITATIONS AND FUTURE ROADMAP

Despite their widespread use, current techniques did not completely solve the intrinsic problem of achieving memory safety in non-memory-safe programming languages. The C programming language and its descendants are widely utilized for compelling economic reasons, including their remarkable compatibility with almost every processor architecture, the generation of efficient and transparent compiled code, their ability to create compact code, adherence to ISO standards, seamless access to hardware, proven reliability in critical systems, as well as comprehensive support from a wide range of tools. Moreover, most of the techniques discussed in this survey were not extensively adopted due to the need for specific software support or due to the performance penalty caused by the additional runtime overhead.

Researchers and experts in the security domain are exploring several directions to solve the issue at its core. In December 2023, the U.S. Cybersecurity and Infrastructure Security Agency (CISA) and other international cybersecurity agencies published *The Case for Memory Safe Roadmaps: Why both C-Suite Executives and Technical Experts Need to Take Memory Safe Coding Seriously* [100], a report specifically designed to address the critical issue of memory safety vulnerabilities in programming languages, encouraging software manufacturers to prioritize the use of memory-safe programming languages. In the report, some hardware-assisted techniques presented in the present survey are strongly suggested as a reasonable trade-off between security and performance. While some of these hardware-based mechanisms are still transitioning from research prototypes to deployed products, CISA and other experts anticipate their significance in an overarching strategy to eliminate memory safety vulnerabilities. Secondly, the usage of memory-safe programming languages is strongly encouraged. Unlike other mitigation strategies that demand ongoing maintenance, such as developing new defenses or sifting through vulnerability scans, using a memory-safe programming language requires no additional effort concerning memory safety once the code is prepared. Until a few years ago, the software development industry lacked a programming language that combined the flexibility and performance of C with built-in memory safety assurances. In 2006, a Mozilla software engineer initiated the development of Rust, a new programming language that achieves memory safety through a unique ownership system where variables follow ownership rules and lifetimes are explicitly managed. This approach effectively eliminates common memory-related issues, such as null pointer dereferences and data races. Additionally, the Rust compiler enforces strict rules at compile time, ensuring that references adhere to a set of safety guarantees, making it a robust choice for systems programming with minimal runtime overhead. Rust version 1.0 was officially released in 2015, gaining widespread adoption by prominent software organizations such as Amazon, Facebook, Google, Microsoft, Mozilla, and other key industry players. Since version 6.1 of the Linux kernel, Rust has been officially integrated into the kernel development [101]. Other researchers, influenced by Rust, also proposed extensions to the C language to incorporate memory safety checks at compile time [102].

B. INFLUENCE OF EMERGING TECHNOLOGIES

1) MACHINE LEARNING

With the rise of deep learning, machine learning algorithms have quickly gained prominence in various domains, including cybersecurity, where the ability to analyze large datasets,

learn recurrent patterns, and make precise predictions can effectively enhance security measures [103], [104], [105], [106], [107], [108]. Typical machine learning techniques adopted in cybersecurity include deep neural networks (DNNs), recurrent neural networks (RNNs), convolutional neural networks (CNNs), deep belief networks (DBNs), and restricted Boltzmann machines (RBMs), often arranged in autoencoder architectures [103], [109]. A well-known application of machine learning in cybersecurity consists in detecting anomalies in network intrusion detection systems (NIDS), where deep learning techniques are employed to monitor a computer network and detect anomalous or spurious traffic, which may indicate the presence of a cyberattack. Such techniques achieve an accuracy level which often surpasses that of traditional signature-based intrusion detection systems [107], [108], [110], [111], [112], [113], [114].

Deep learning techniques have also been widely adopted in system-level malware detection to accompany or potentially supersede traditional signature-based methods [104], [105], [106], [115], [116], [117]. In the context of memory integrity techniques, recent investigations have focused on applying machine learning techniques to mitigate return-oriented programming and achieve control flow integrity. On this front, Pfaff et al. [118] introduced HadROP, a ROP attack detection technique leveraging a support vector machine (SVM) learning method trained on statistical data extracted from hardware performance counters. Elsabagh et al. [119] presented EigenROP, an unsupervised anomaly detection mechanism to defend against ROP attacks that uses microarchitecture-agnostic program features (e.g., memory locality, register traffic, memory reuse distance) and leverages a dynamic instrumentation framework (Intel Pin). Li et al. [120] later presented ROPNN, which surpasses the performance of EigenROP using a CNN-based classifier with minimal runtime overhead and is also effective against attack patterns not represented in the training dataset. DeepCheck [121] is a CFI technique to classify Intel ISA execution traces by extracting relevant execution states using the Intel Processor Trace (IPT) performance analysis feature and processing them with a DNN classifier. Although the classifier used in DeepCheck is trained on data extracted from the Control Flow Graphs (CFGs) of numerous programs, CFG information is not required for classification at runtime. Like DeepCheck, HeNet [122] is a CFI technique that leverages IPT to analyze the execution state of a program, but adopts a hierarchical ensemble of DNNs to enhance ROP detection accuracy. More recently, Koranek et al. [123] developed specialized LSTM models to analyze RISC-V ISA execution traces and determine whether they were subject to ROP exploitation.

Security-oriented program analysis is another significant branch of security research leveraging machine learning applications. In this domain, static program analyzers are enhanced with machine learning techniques to detect and analyze vulnerabilities related to control flow and data flow [104], [124], [125], [126], [127].

A significant challenge in adopting machine-learning techniques in cybersecurity is the generation of relevant datasets to train, validate, and test deep neural networks. In fact, in order to be effective, such data should be compatible with the selected learning method in terms of learnable features and should also be representative of a large number of possible attack patterns to enable a model to generalize [104]. Therefore, investigating suitable techniques to generate relevant datasets is crucial to most of the works discussed above.

2) QUANTUM COMPUTING

Quantum computing is another emerging technology that is having a strong impact on the domain of cyber-security. Traditional cryptographic protocols (e.g., public-key cryptosystems such as RSA [128]) often rely on the assumption that traditional computers cannot efficiently solve complex computing problems such as factorization of large prime numbers. However, a number of quantum algorithms were devised to efficiently solve such problems when executed on a large enough quantum computer. For instance, Shor's algorithm [129] can quickly solve integer factorization, thus easily breaking the RSA cryptosystem. The potential prospect of powerful quantum computers becoming widely available has led to the need for developing quantum-resistant cryptographic protocols, in what is known as *post-quantum cryptography* [130], [131]. When considering the specific domain of memory integrity, which is the subject of this survey, the impact of quantum algorithms mostly pertains to the requirement of ensuring that any defense technique utilizing encryption or hashing as part of its specification (e.g., ARM Pointer Authentication [53]) is updated to employ post-quantum cryptographic protocols [130], [131].

VI. CONCLUSION

This paper presented a survey on memory safety techniques for memory-unsafe languages ranging from older mechanisms to state-of-the-art solutions, thus providing a comprehensive overview of the advancements in the field during the last twenty years. A comparative analysis of the investigated techniques was presented to assess their applicability, identifying areas where further research and improvements are required. Furthermore, a roadmap for memory safety in the context of future research was also provided, highlighting current challenges and emerging trends. Overall, this survey highlighted that memory corruption vulnerabilities still persist in modern software and constitute a serious security issue after years of specialized research. Each specific software product may require a unique investment strategy to address the vulnerabilities related to memory-unsafe code and minimize the related security risks, at the cost of a potential decrease in performance. While there is no universal solution to enhance the security of modern software systems, it is crucial for software developers and

hardware manufacturers to be aware of the related problems and properly address them by employing the most recent security countermeasures and development strategies.

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*. Berlin, Germany: Springer, 2012, pp. 86–106.
[2] Qualys Security Advisory, *PWNKIT: Local Privilege Escalation in Polkit's PKEXEC (CVE-2021–4034)*, Qualys, Foster City, CA, USA, 2022.
[3] J. P. Anderson, "Computer security technology planning study," Deputy Command Manag. Syst., Electron. Syst. Division, Bedford, MS, USA, Tech. Rep. ESD-TR-73-51, 1972.
[4] E. H. Spafford, "The internet worm program: An analysis," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 19, no. 1, pp. 17–57, Jan. 1989.
[5] S. Chasin, "Bugtraq mailing list," to be published.
[6] G. Lyon, "Fulldisclosure—Improving network security through full disclosure," to be published.
[7] E. Levy, "Smashing the stack for fun and profit," 1996.
[8] C. Sun, V. Le, Q. Zhang, and Z. Su, "Toward understanding compiler bugs in GCC and LLVM," in *Proc. 25th Int. Symp. Softw. Test. Anal.*, New York, NY, USA, Jul. 2016, pp. 294–305.
[9] V. D'Silva, M. Payer, and D. Song, "The correctness-security gap in compiler optimization," in *Proc. IEEE Secur. Privacy Workshops*, May 2015, pp. 73–87.
[10] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama, "Towards optimization-safe systems: Analyzing the impact of undefined behavior," in *Proc. 24th ACM Symp. Operating Syst. Princ.*, New York, NY, USA, Nov. 2013, pp. 260–275.
[11] Z. Yang, B. Johannesmeyer, A. T. Olesen, S. Lerner, and K. Levchenko, "Dead store elimination (still) considered harmful," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 1025–1040.
[12] (2023). *Erasing Sensitive Data—String and Array Utilities—The GNU C Library Manual*. Accessed: Jan. 23, 2023. [Online]. Available: https://www.gnu.org/software/libc/manual/html_node/Erasing-Sensitive-Data.html
[13] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, and J. A. Halderman, "The matter of heartbleed," in *Proc. 2014 Conf. Internet Meas. Conf.*, New York, NY, USA, 2014, pp. 475–488.
[14] (2021). *Buffer Overflow and Underflow in Getcwd()—Glibc*. Accessed: Jan. 23, 2023. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2021-3999
[15] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 13, no. 1, pp. 1–40, Nov. 2009.
[16] J. Pincus and B. Baker, "Beyond stack smashing: Recent advances in exploiting buffer overruns," *IEEE Secur. Privacy*, vol. 2, no. 4, pp. 20–27, Jul. 2004.
[17] (2022). *CWE-119: Improper Restriction of Operations Within the Bounds of a Memory Buffer*. Accessed: May 22, 2022. [Online]. Available: https://cwe.mitre.org/data/definitions/119.html
[18] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the expressiveness of return-into-libc attacks," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*. Berlin, Germany: Springer, 2011, pp. 121–141.
[19] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, pp. 1–34, Mar. 2012.
[20] A. Peslyak, "LPR libc return exploit," Solar Designer, 1997.
[21] E. Barberis, P. Frigo, M. Muench, H. Bos, and C. Giuffrida, "Branch history injection: On the effectiveness of hardware mitigations against cross-privilege Spectre-v2 attacks," in *Proc. 31st USENIX Secur. Symp.*, Boston, MA, USA, Aug. 2022, pp. 971–988.
[22] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, "A survey on cache management mechanisms for real-time embedded systems," *ACM Comput. Surv.*, vol. 48, no. 2, pp. 1–36, Nov. 2015.
[23] M. H. Islam Chowdhuryy, H. Liu, and F. Yao, "BranchSpec: Information leakage attacks exploiting speculative branch instruction executions," in *Proc. IEEE 38th Int. Conf. Comput. Design (ICCD)*, Oct. 2020, pp. 529–536.
[24] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proc. 7th USENIX Secur. Symp.*, vol. 98, Jan. 1998, p. 5.
[25] H. Etoh, "GCC extension for protecting applications from stack-smashing attacks," IBM, Armonk, NY, USA, Tech. Rep., Jan. 2004.
[26] R. Henderson, "Reimplementation of ibm stack-smashing protector," Red Hat, Raleigh, NC, USA, 2005.
[27] GCC Team, *GCC 4.1 Release Series Changes, New Features, and Fixes*, Free Softw. Found. Inc, Boston, MA, USA, 2005.
[28] H. Shen, "Add a new option '-fstack-protector-strong,'" Google LLC, Menlo Park, CA, USA, 2012.
[29] T. H. Y. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *Proc. 10th ACM Symp. Inf., Comput. Commun. Secur.*, New York, NY, USA, Apr. 2015, pp. 555–566.
[30] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Address-Sanitizer: A fast address sanity checker," in *Proc. USENIX Annu. Tech. Conf.*, Boston, MA, USA, 2012, pp. 309–318.
[31] LLVM Team, *LLVM 3.1 Release Notes*, LLVM Found., Los Altos, CA, USA, 2012.
[32] GCC Team, *GCC 4.8 Release Changes*, Free Softw. Found. Inc, Boston, MA, USA, 2014.
[33] GCC Team, *Heap Consistency Checking*, Free Softw. Found. Inc, Boston, MA, USA, 2003.
[34] M. Eckert, A. Bianchi, R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "HeapHopper: Bringing bounded model checking to heap implementation security," in *Proc. 27th USENIX Conf. Secur. Symp.*, 2018, pp. 99–116.
[35] C. Cowan, S. Beattie, J. E. Johansen, and P. Wagle, "PointGuard: Protecting pointers from buffer overflow vulnerabilities," in *Proc. USENIX Secur. Symp.*, 2003, pp. 91–104.
[36] *Platform Independent Code Obfuscation*. Accessed: Oct. 22, 2023. [Online]. Available: https://www.divaportal.org/smash/get/diva2:699631/FULLTEXT01.pdf
[37] B. Anckaert, M. Jakubowski, R. Venkatesan, and N. Saw, "Practical data location obfuscation," Microsoft Res., Redmond, WA, USA, Tech. Rep. MSR-TR-2009-3, Jan. 2009.
[38] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits," in *Proc. 12th USENIX Secur. Symp.*, Washington, DC, USA, Aug. 2003, pp. 1–17.
[39] T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji, "Software obfuscation on a theoretical basis and its implementation," *IEICE Trans. Fundam. Electron., Commun. Comput. Sci.*, vol. 86, no. 1, pp. 176–186, 2003.
[40] H. Chen, L. Yuan, X. Wu, B. Zang, B. Huang, and P.-C. Yew, "Control flow obfuscation with information flow tracking," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, New York, NY, USA, Dec. 2009, pp. 391–400.
[41] J. Jelinek, "Object size checking to prevent (some) buffer overflows," Red Hat, Raleigh, NC, USA, 2004.
[42] S. Poyarekar, "Broadening compiler checks for buffer overflows in fortify_source," Red Hat, Raleigh, NC, USA, 2021.
[43] E. Pilkington, "A new builtin: Builtin_dynamic_object_size," Independ. Res., 2019.
[44] Y. Han, P. Shah, V. Nguyen, L. Ma, and R. Livingston, "OpenOSC: Open source object size checking library with built-in metrics," in *Proc. IEEE Cybersecurity Develop. (SecDev)*, Sep. 2019, p. 143.
[45] The Clang Team, *Shadow Call Stack*, LLVM Found., Los Altos, CA, USA, 2019.
[46] P. Turner, "Retpoline: A software construct for preventing branch-target-injection," Google LLC, Menlo Park, CA, USA, 2018.
[47] *Retpoline: A Branch Target Injection Mitigation*, Intel, Santa Clara, CA, USA, 2018.
[48] *Retpoline: A Software Construct for Preventing Branch-Target-Injection*. Accessed: Aug. 21, 2023. [Online]. Available: https://support.google.com/faqs/answer/7625886
[49] PaX Team, *PaX: Address Space Layout Randomization*, Open Source Secur., Inc, Lancaster, PA, USA, 2003.
[50] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over ASLR: Attacking branch predictors to bypass ASLR," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2016, pp. 1–13.
[51] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the line: Practical cache attacks on the MMU," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, San Diego, CA, USA, 2017, p. 26.

[52] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space ASLR," in *Proc. IEEE Symp. Secur. Privacy*, May 2013, pp. 191–205.

[53] *Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions*, Qualcomm Technologies, San Diego, CA, USA, 2017.

[54] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan, "PAC it up: Towards pointer integrity using ARM pointer authentication," in *Proc. 28th USENIX Conf. Secur. Symp.*, 2019, pp. 177–194.

[55] S. Yoo, J. Park, S. Kim, Y. Kim, and T. Kim, "In-kernel control-flow integrity on commodity OSes using ARM pointer authentication," in *Proc. USENIX Secur. Symp.*, 2021, pp. 89–106.

[56] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, "PACMAN: Attacking ARM pointer authentication with speculative execution," in *Proc. 49th Annu. Int. Symp. Comput. Archit.*, New York, NY, USA, Jun. 2022, pp. 685–698.

[57] G. Serra, P. Fara, G. Cicero, F. Restuccia, and A. Biondi, "PAC-PL: Enabling control-flow integrity with pointer authentication in FPGA SoC platforms," in *Proc. IEEE 28th Real-Time Embedded Technol. Appl. Symp. (RTAS)*, Giorgiomaria Cicero, France, May 2022, pp. 241–253.

[58] *ARM A-Profile Architecture Developments 2018: ARMV8.5-A*, ARM, Cambridge, U.K., 2018.

[59] *ARMV8-A Reference Manual*, ARM, Cambridge, U.K., 2023.

[60] *Intel 64 and IA-32 Architectures Software Developer's Manual*, Intel, Santa Clara, CA, USA, 2023.

[61] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov, "Memory tagging and how it improves C/C++ memory safety," Google LLC, Menlo Park, CA, USA, 2018.

[62] *The ARM64 Memory Tagging Extension in Linux*. Accessed: May 1, 2023. [Online]. Available: https://lwn.net/Articles/834289/

[63] *ARMV8.5-A Memory Tagging Extension*, ARM, Cambridge, U.K., 2019.

[64] M. Unterguggenberger, D. Schrammel, P. Nasahl, R. Schilling, L. Lamster, and S. Mangard, "Multi-tag: A hardware–software co-design for memory safety based on multi-granular memory tagging," in *Proc. ACM Asia Conf. Comput. Commun. Secur.*, Jul. 2023, pp. 177–189.

[65] R. Ramakesavan, D. Zimmerman, P. Singaravelu, G. Kuan, B. Vajda, S. Gibbons, and G. Beeraka, "Intel memory protection extensions enabling guide," Intel, Santa Clara, CA, USA, Tech. Rep. 751866, 2015.

[66] GCC Team, *Intel® Memory Protection Extensions (Intel® MPX) Support in the GCC Compiler*, Free Softw. Found. Inc, Boston, MA, USA, 2016.

[67] The Linux Kernel Documentation, *Intel(R) Memory Protection Extensions (MPX)*, Linux Found., San Francisco, CA, USA, 2016.

[68] *Support for Intel® Memory Protection Extensions (Intel® MPX) Technology*, Intel, Santa Clara, CA, USA, 2019.

[69] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel MPX explained: A cross-layer analysis of the Intel MPX system stack," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 2, no. 2, pp. 1–30, Jun. 2018.

[70] *Control-Flow Enforcement Technology Specification*, Intel, Santa Clara, CA, USA, 2019.

[71] S. Tauner and M. Telesklav, "Comparative analysis and enhancement of CFG-based hardware-assisted CFI schemes," *ACM Trans. Embedded Comput. Syst.*, vol. 20, no. 5s, pp. 1–25, Sep. 2021.

[72] V. Shanbhogue, D. Gupta, and R. Sahita, "Security analysis of processor instruction set architecture for enforcing control-flow integrity," in *Proc. 8th Int. Workshop Hardw. Architectural Support Secur. Privacy*, Jun. 2019, pp. 1–11.

[73] M. Kucab, P. Borylo, and P. Cholda, "Performance impact of control flow enforcement technology (CET)," in *Proc. 25th Conf. Innov. Clouds, Internet Netw. (ICIN)*, Mar. 2022, pp. 96–100.

[74] *Intel Architecture Instruction Set Extensions and Future Features*, Intel, Santa Clara, CA, USA, 2023.

[75] R. N. M. Watson, S. W. Moore, P. Sewell, and P. G. Neumann, "An introduction to cheri," Univ. Cambridge, Cambridge, U.K., Tech. Rep. UCAM-CL-TR-941, 2019.

[76] *CHERI C/C++ Programming Guide*. Accessed: Jul. 15, 2023. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-947.pdf

[77] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera, "CHERI: A hybrid capability-system architecture for scalable software compartmentalization," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 20–37.

[78] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI capability model: Revisiting RISC in an age of risk," in *Proc. ACM/IEEE 41st Int. Symp. Comput. Archit. (ISCA)*, Jun. 2014, pp. 457–468.

[79] *ARM Virtual Address Tagging*. Accessed: Aug. 18, 2023. [Online]. Available: https://developer.arm.com/documentation/den0024/a/The-Memory-Management-Unit/Translation-table-configuration/Virtual-Address-tagging3

[80] *LLVM Hardware-Assisted AddressSanitizer Design Documentation*. Accessed: Oct. 18, 2023. [Online]. Available: https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html

[81] *Understanding HWASan Reports*. Accessed: Oct. 22, 2023. [Online]. Available: https://source.android.com/docs/security/test/memory-safety/hwasan-reports

[82] *HWAddressSanitizer—Android*. Accessed: Oct. 22, 2023. [Online]. Available: https://source.android.com/docs/security/test/hwasan

[83] A. Partap and D. Boneh, "Memory tagging: A memory efficient design," 2022, *arXiv:2209.00307*.

[84] P. E. Black, "SARD: A software assurance reference dataset," in *Anonymous Cybersecurity Innovation Forum*. Cybersecurity Innovation Forum, Federal Business Council, 2017.

[85] *Juliet Test Suite V1.2 for C/C++*. NSA Center for Assured Software. Accessed: Mar. 20, 2024. [Online]. Available: https://samate.nist.gov/SARD/downloads/documents/Juliet_Test_Suite_v1.2_for_C_Cpp_-_User_Guide.pdf

[86] IARPA: Intelligence Advanced Research Projects Activity, *STONESOUP—Securely Taking On New Executable Software of Uncertain Provenance*, Office Director Nat. Intell. (ODNI), Washington, DC, USA, 2009.

[87] A. Benameur, N. S. Evans, and M. C. Elder, "MINESTRONE: Testing the SOUP," in *Proc. CSET*, 2013, pp. 1–8.

[88] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "RIPE: Runtime intrusion prevention evaluator," in *Proc. 27th Annu. Comput. Secur. Appl. Conf.*, Dec. 2011, pp. 41–50.

[89] 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*. New York, NY, USA: ACM, 2018, pp. 81–116.

[90] M. Zhang and R. Sekar, "Control flow and code integrity for COTS binaries: An effective defense against real-world ROP attacks," in *Proc. 31st Annu. Comput. Secur. Appl. Conf.*, 2015, pp. 91–100.

[91] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, "SGX-shield: Enabling address space layout randomization for SGX programs," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2017, pp. 1–15.

[92] J. C. Detter and R. Mutschlechner, "Performance and entropy of various aslr implementations," Dept. Comput. Sci., Univ. Wisconsin-Madison, Madison, WI, USA, Tech. Rep., 2015.

[93] S. Zhou and J. Chen, "Experimental evaluation of the defense capability of ARM-based systems against buffer overflow attacks in wireless networks," in *Proc. IEEE 10th Int. Conf. Electron. Inf. Emergency Commun. (ICEIEC)*, Jul. 2020, pp. 375–378.

[94] B. M. Calatayud and L. Meany, "A comparative analysis of buffer overflow vulnerabilities in high-end IoT devices," in *Proc. IEEE 12th Annu. Comput. Commun. Workshop Conf. (CCWC)*, Jan. 2022, pp. 0694–0701.

[95] Y. Wang, J. Wu, T. Yue, Z. Ning, and F. Zhang, "RetTag: Hardware-assisted return address integrity on RISC-V," in *Proc. 15th Eur. Workshop Syst. Secur.*, New York, NY, USA, Apr. 2022, pp. 50–56.

[96] G. Roascio, G. Serra, and V. Eftekhari Moghadam, "Em-RIPE: Runtime intrusion prevention evaluator for ARM microcontroller systems," in *Proc. Int. Conf. Electr., Comput., Commun. Mechatronics Eng. (ICECCME)*, Nov. 2022, pp. 1–6.

[97] G. Serra, S. Di Leonardi, and A. Biondi, "X-RIPE: A modern cross-platform runtime intrusion prevention evaluator," in *Proc. 17th Annu. Workshop Operating Syst. Platforms Embedded Real-Time Appl. (OSPERT)*, 2022, pp. 49–55.

[98] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *Proc. IEEE Symp. Secur. Privacy*, May 2013, pp. 48–62.

[99] N. R. Kisore, "A qualitative framework for evaluating buffer overflow protection mechanisms," *Int. J. Inf. Comput. Secur.*, vol. 8, no. 3, pp. 272–307, 2016.

[100] CISA: Cybersecurity Infrastructure Security Agency. *The Case for Memory Safe Roadmaps*. Accessed: Mar. 20, 2024. [Online]. Available: https://www.cisa.gov/sites/default/files/2023-12/The-Case-for-Memory-Safe-Roadmaps-508c.pdf

[101] Kees Cook. *Rust Introduction for V6.1-rc1*. Accessed: Mar. 20, 2024. [Online]. Available: https://lore.kernel.org/lkml/202210010816.1317F2C@keescook/
[102] R. Bagnara, A. Bagnara, and F. Serafini, "C-rusted: The advantages of rust, in C, without the disadvantages," 2023, *arXiv:2302.05331*.
[103] D. Berman, A. Buczak, J. Chavis, and C. Corbett, "A survey of deep learning methods for cyber security," *Information*, vol. 10, no. 4, p. 122, Apr. 2019.
[104] Y.-H. Choi, P. Liu, Z. Shang, H. Wang, Z. Wang, L. Zhang, J. Zhou, and Q. Zou, "Using deep learning to solve computer security challenges: A survey," *Cybersecurity*, vol. 3, no. 1, pp. 1–32, Dec. 2020.
[105] J. Qiu, J. Zhang, W. Luo, L. Pan, S. Nepal, and Y. Xiang, "A survey of Android malware detection with deep neural models," *ACM Comput. Surv.*, vol. 53, no. 6, pp. 1–36, Nov. 2021.
[106] K. Liu, S. Xu, G. Xu, M. Zhang, D. Sun, and H. Liu, "A review of Android malware detection approaches based on machine learning," *IEEE Access*, vol. 8, pp. 124579–124607, 2020.
[107] Z. Ahmad, A. Shahid Khan, C. Wai Shiang, J. Abdullah, and F. Ahmad, "Network intrusion detection system: A systematic study of machine learning and deep learning approaches," *Trans. Emerg. Telecommun. Technol.*, vol. 32, no. 1, p. e4150, Jan. 2021.
[108] J. Lansky, S. Ali, M. Mohammadi, M. K. Majeed, S. H. T. Karim, S. Rashidi, M. Hosseinzadeh, and A. M. Rahmani, "Deep learning-based intrusion detection systems: A systematic review," *IEEE Access*, vol. 9, pp. 101574–101599, 2021.
[109] K. Shaukat, S. Luo, V. Varadharajan, I. A. Hameed, and M. Xu, "A survey on machine learning techniques for cyber security in the last decade," *IEEE Access*, vol. 8, pp. 222310–222354, 2020.
[110] Y. Chen, Y. Li, X.-Q. Cheng, and L. Guo, "Survey and taxonomy of feature selection algorithms in intrusion detection system," in *Information Security and Cryptology*, H. Lipmaa, M. Yung, and D. Lin, Eds. Berlin, Germany: Springer, 2006, pp. 153–167.
[111] G. Kathareios, A. Anghel, A. Mate, R. Clauberg, and M. Gusat, "Catch it if you can: Real-time network anomaly detection with low false alarm rates," in *Proc. 16th IEEE Int. Conf. Mach. Learn. Appl. (ICMLA)*, Dec. 2017, pp. 924–929.
[112] M. Yousefi-Azar, V. Varadharajan, L. Hamey, and U. Tupakula, "Autoencoder-based feature learning for cyber security applications," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, May 2017, pp. 3854–3861.
[113] Y. Mirsky, T. Doitshman, Y. Elovici, and A. Shabtai, "Kitsune: An ensemble of autoencoders for online network intrusion detection," 2018, *arXiv:1802.09089*.
[114] N. Borgioli, L. Thi Xuan Phan, F. Aromolo, A. Biondi, and G. Buttazzo, "Real-time packet-based intrusion detection on edge devices," in *Proc. Cyber-Physical Syst. Internet Things Week*, New York, NY, USA, May 2023, pp. 234–240.
[115] I. Firdausi, C. Lim, A. Erwin, and A. S. Nugroho, "Analysis of machine learning techniques used in behavior-based malware detection," in *Proc. 2nd Int. Conf. Adv. Comput., Control, Telecommun. Technol.*, Dec. 2010, pp. 201–203.
[116] J. Sahs and L. Khan, "A machine learning approach to Android malware detection," in *Proc. Eur. Intell. Secur. Informat. Conf.*, Aug. 2012, pp. 141–147.
[117] U.-E.-H. Tayyab, F. B. Khan, M. H. Durad, A. Khan, and Y. S. Lee, "A survey of the recent trends in deep learning based malware detection," *J. Cybersecurity Privacy*, vol. 2, no. 4, pp. 800–829, Sep. 2022.
[118] D. Pfaff, S. Hack, and C. Hammer, "Learning how to prevent return-oriented programming efficiently," in *Engineering Secure Software and Systems*, Milan, Italy. Cham, Switzerland: Springer, 2015, pp. 68–85.
[119] M. Elsabagh, D. Barbara, D. Fleck, and A. Stavrou, "Detecting ROP with statistical learning of program characteristics," in *Proc. 7th ACM Conf. Data Appl. Secur. Privacy*, Mar. 2017, pp. 219–226.
[120] X. Li, Z. Hu, H. Wang, Y. Fu, P. Chen, M. Zhu, and P. Liu, "ROPNN: Detection of ROP payloads using deep neural networks," 2018, *arXiv:1807.11110*.
[121] J. Zhang, W. Chen, and Y. Niu, "DeepCheck: A non-intrusive control-flow integrity checking based on deep learning," 2019, *arXiv:1905.01858*.
[122] L. Chen, S. Sultana, and R. Sahita, "HeNet: A deep learning approach on Intel processor trace for effective exploit detection," in *Proc. IEEE Secur. Privacy Workshops (SPW)*, May 2018, pp. 109–115.
[123] D. F. Koranek, S. R. Graham, B. J. Borghetti, and W. C. Henry, "Identification of return-oriented programming attacks using RISC-V instruction trace data," *IEEE Access*, vol. 10, pp. 45347–45364, 2022.
[124] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *Proc. 24th USENIX Secur. Symp.*, 2015, pp. 611–626.
[125] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang, "Neural nets can learn function type signatures from binaries," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 99–116.
[126] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 363–376.
[127] W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and X. Xing, "LEMNA: Explaining deep learning based security applications," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2018, pp. 364–379.
[128] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 26, no. 1, pp. 96–99, Jan. 1983.
[129] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proc. 35th Annu. Symp. Found. Comput. Sci.*, 1994, pp. 124–134.
[130] D. J. Bernstein and L. Tanja, "Post-quantum cryptography," *Nature*, vol. 549, no. 7671, pp. 188–194, 7671.
[131] L. Chen, L. Chen, S. Jordan, Y.-K. Liu, D. Moody, R. Peralta, R. A. Perlner, and D. Smith-Tone, "Report on post-quantum cryptography, volume 12," US Dept. Commerce, Nat. Inst. Standards Technol., Gaithersburg, MD, USA, Tech. Rep. NIST IR 8105, 2016.

VAHID EFTEKHARI MOGHADAM is a Ph.D. student at the Department of Computer & Control Engineering (DAUIN) at Politecnico di Torino (PoliTO), located in Turin, Italy. He works as a security researcher in the Hardware & Embedded Security Group (RHESGroup) under the supervision of Prof. Paolo Prinetto. He received his M.Sc. in computer engineering (Embedded Systems) from PoliTO in 2020. Thenceforth he joined the joint collaboration between TIM S.p.A. (Telecom Italia) Ph.D. Academy & PoliTO, where he's working on the security vulnerabilities of embedded devices targeting architectural solutions to improve their resilience and vulnerability tolerance.

GABRIELE SERRA (Member, IEEE) received the joint M.Sc. degree in embedded computing systems from Università di Pisa and Scuola Superiore Sant'Anna, Pisa, Italy, and the Ph.D. degree in embedded systems from Scuola Superiore Sant'Anna. He is a Postdoctoral Researcher with Scuola Superiore Sant'Anna. He is with the Real-Time System Laboratory (ReTiS), under the supervision of Prof. Giorgio Buttazzo. In May 2019, he started working on an ongoing project in partnership with Rete Ferroviaria Italiana, targeting the design of a safety-critical real-time operating systems to be used in the railway scenario.

FEDERICO AROMOLO received the M.Sc. degree in embedded computing systems from Scuola Superiore Sant'Anna, Pisa, and the University of Pisa, and the Ph.D. degree in computer engineering from Scuola Superiore Sant'Anna. He is an Assistant Professor with the Real-Time Systems (ReTiS) Laboratory, Scuola Superiore Sant'Anna. His research interests include real-time scheduling algorithms, schedulability analysis, real-time operating systems, and cyber-security for embedded systems.

GIORGIO BUTTAZZO (Member, IEEE) received the degree in electronic engineering from the University of Pisa, the M.S. degree in computer science from the University of Pennsylvania, and the Ph.D. degree in computer engineering from Scuola Superiore Sant'Anna, Pisa. He is a Full Professor of computer engineering with Scuola Superiore Sant'Anna. He has authored seven books on real-time systems and more than 300 papers in the field of real-time systems, robotics, and neural networks. He received 11 best paper awards. He is the Editor-in-Chief of *Real-Time Systems* and an Associate Editor of the *ACM Transactions on Cyber-Physical Systems*.

PAOLO PRINETTO (Senior Member, IEEE) received the M.S. degree in electronic engineering from Politecnico di Torino, Italy, in 1997. He is a Full Professor of computer engineering with Politecnico di Torino (50%) and the IMT–Institute for Advanced Studies Lucca, Italy, (50%). He is also the Director of the CINI–Cybersecurity National Laboratory; and a Coordinator of the Programs CyberChallenge, OliCyber, and Cyber-HighSchools. His research activities are mainly focused on hardware security, digital systems design and test, system dependability. From 2010 to 2014, he served as an Appointed Member for the Scientific Committee of the French "Centre National de la Recherche Scientifique" (CNRS). In 2012, he was honored of the title "Doctor Honoris Causa" of the Technical University of Cluj-Napoca, Romania. He is serving as an Italian Representative with the Civil Security for Society Sub-Group of the EU Shadow Strategic Program Committee of Horizon Europe–the Framework Program for Research and Innovation. From 2013 to 2019, he was the President of CINI (Italian National Inter University Consortium for Informatics). From 2013 to 2019, he was the Vice-Chair of the International Federation for Information Processing (IFIP) Technical Committee TC 10–Computer Systems Technology. In 2000 and 2003, he served as the Elected Chair for the IEEE Computer Society Test Technology Technical Council (TTTC).

Open Access funding provided by 'Politecnico di Torino' within the CRUI CARE Agreement