Artifact Evaluated NDSS Available Functional

Retrofitting XoM for Stripped Binaries without Embedded Data Relocation

Chenke Luo†, Jiang Ming‡, Mengfei Xie†, Guojun Peng† and Jianming Fu†*
†Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education,
School of Cyber Science and Engineering, Wuhan University
‡Department of Computer Science, School of Science and Engineering, Tulane University
Email: kernelthread@whu.edu.cn, jming@tulane.edu, {mfxie96, guojpeng, jmfu}@whu.edu.cn
* Jianming Fu is the corresponding author.

***Abstract*—System programs are frequently coded in memory-unsafe languages such as C/C++, rendering them susceptible to a variety of memory corruption attacks. Among these, just-in-time return-oriented programming (JIT-ROP) stands out as an advanced form of code-reuse attack designed to circumvent code randomization defenses. JIT-ROP leverages memory disclosure vulnerabilities to dynamically harvest reusable code gadgets and construct attack payloads in real-time. To counteract JIT-ROP threats, researchers have developed multiple execute-only memory (XoM) prototypes to prevent dynamic reading and disassembly of memory pages. XoM, akin to the widely deployed W⊕X protection, holds promise in enhancing security. However, existing XoM solutions may not be compatible with legacy and commercial off-the-shelf (COTS) programs, or they may require patching the protected binary to separate code and data areas, leading to poor reliability. In addition, some XoM methods have to modify the underlying architectural mechanism, compromising compatibility and performance.**

In this paper, we present *PXoM*, a practical technique to seamlessly retrofit XoM into stripped binaries on the x86-64 platform. As handling the mixture of code and data is a well-known challenge for XoM, most existing methods require the strict separation of code and data areas via either compile-time transformation or binary patching, so that the unreadable permission can be safely enforced at the granularity of memory pages. In contrast to previous approaches, we provide a fine-grained memory permission control mechanism to restrict the read permission of code while allowing legitimate data reads within code pages. This novelty enables PXoM to harden stripped binaries but without resorting to error-prone embedded data relocation. We leverage Intel's hardware feature, Memory Protection Keys, to offer an efficient fine-grained permission control. We measure PXoM's performance with both micro- and macro-benchmarks, and it only introduces negligible runtime overhead. Our security evaluation shows that PXoM leaves adversaries with little wiggle room to harvest all of the required gadgets, suggesting PXoM is practical for real-world deployment.

I. Introduction

The perpetual competition between cyber adversaries and defenders on memory corruption vulnerabilities has intensified, resulting in an ongoing struggle [1]–[6]. The prevalence of W⊕X protection (i.e., memory cannot be writable and executable at the same time) in modern operating systems has led attackers to reuse code snippets from the vulnerable program to construct attacks. Adversaries identify these code snippets, also known as "gadgets," by examining the disassembled binary code [7]. Subsequently, they connect these gadgets in a precise sequence to create harmful payloads and redirect the control flow to the gadgets to launch the attack. To mitigate this threat, researchers have proposed various code randomization techniques [8]–[19] to impede the construction of gadgets by reorganizing the code layout in memory. However, code randomization is susceptible to memory disclosure, which makes the randomized code layout evident to attackers and undermines the fundamental memory secrecy assumption of code randomization [20]. The technique of JIT-ROP [21] leverages repeated exploitation of memory disclosure vulnerabilities to collect code gadgets on-the-fly. This is accomplished by utilizing the leaked code pointers present on memory pages. Consequently, JIT-ROP can circumvent code randomization protection, even rendering fine-grained randomization strategies ineffective [22]. The premise of JIT-ROP relies on the disclosure of memory pages, where attackers must first traverse disassembled code to gather the required gadgets for the payload construction. Therefore, a common JIT-ROP defense is to enforce a fine-grained memory permission policy to restrict arbitrary read to code pages.

Execute-only memory (XoM) [23]–[29] has emerged as a prominent defense against memory disclosure. By revoking the read privilege of executable memory, XoM deprives attackers of the ability to inspect the code after code randomization has been applied. XoM implementations have utilized software emulation [23], [24], [26], [29] or hardware features [25], [27], [28] to achieve this objective. Unfortunately, existing XoM prototypes have failed to gain popularity, and one of the major obstacle comes from the false alarms caused by legal data-in-code reads. Ideally, if code and data areas are strictly separated, XoM can safely remove the read privilege only from code sections. However, for optimization purposes, code-data mixture cases are not rare. For example, compilers may emit data near their accessing code to exploit spatial locality [30].

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240825
www.ndss-symposium.org

XnR [23] is the first approach to leverage XoM to defend against JIT-ROP attacks, based on the assumption that no data is embedded in the code segment. Subsequent XoM papers have attempted to address code-data separation in two ways. The first class of work explicitly separates code and data areas through custom compilers and linkers [24]–[26]. Obviously, they cannot protect a large number of legacy and COTS binaries. The second class of XoM work attempts to harden binary code [27]–[29]. Nonetheless, they either rely on debug symbols or error-prone binary patching to differentiate embedded data from code, making these approaches impractical. In particular, HideM [27] modifies the architectural mechanism by segregating all data and code into separate caches. This cache mode change has a negative impact on performance and compatibility, as modern CPUs no longer have separate code and data caches. These limitations necessitate further research in restricting adversaries' ability to exploit memory disclosure. On the other hand, Destructive Code Reads (DCR) [31], [32] can tolerate code disclosure by destroying the disclosed code immediately after it is read, thus preventing its execution. DCR addresses the challenge of handling legitimate data reads within code pages, thereby offering enhanced compatibility for protecting binaries. However, the security guarantees of DCR have been compromised by code inference attacks [33].

This paper contributes to the ongoing research in XoM policy enforcement by presenting a novel technique called *PXoM*. Our approach safeguards stripped binaries from JIT-ROP attacks on the x86-64 platform without the need for embedded data relocation. The core of PXoM lies in an efficient and fine-grained memory access control policy, which assigns the R⊕X permission to different blocks within a memory page. This approach is in contrast to the previous method that required patching of the protected binary [28], which involved relocating embedded data out of code pages and updating code-to-data references. We note that binary rewriting for relocating embedded data remains a nascent technique, as highlighted in the latest study [34]. Our technique enables legitimate data-in-code reads by enforcing the execute-only permission on code areas only, rather than at the granularity of the whole memory page. We take advantage of a Intel hardware feature, Memory Protection Keys (MPK) [35], [36], to regulate read requests to code areas and embedded data areas at the kernel level, thus minimizing the performance overhead of our approach.

Specifically, to bypass the barrier of precise binary disassembly [37], [38], we propose a *Unidirectional Disassembly* strategy, which is able to identify all data embedded in code areas without false negatives. We customize the binary loader in Linux kernel to load the PXoM-protected binaries, and implement a runtime monitor in kernel to dynamically scrutinize all read requests to code pages. To further enhance PXoM's performance on frequently accessed embedded data, we have developed a cache-like optimization policy. Our secure evaluation measures the adversaries' ability to launch a ROP attack. Our results have revealed a minimal presence of gadgets in the PXoM-protected binaries, and these leftover gadgets are far from being sufficient to construct a harmful payload. We conduct a multifaceted performance evaluation with microbenchmarks, marcobenchmarks, and real-world applications, including lmbench [39], SPEC CPU 2006 & 2017 [40], [41], three web servers, and four database software. The results show that PXoM only incurs negligible runtime overhead, ranging from 0.22% to 0.82% on average.

In a nutshell, we make the following key contributions:

- We propose a new hardware-assisted XoM technique, PXoM, which hardens stripped binaries to impede memory disclosure attempts and eventually prevent JIT-ROP attacks. Our work is an advancement in the utilization of hardware features for systems security.
- Our novel fine-grained memory access control policy enable us to overcome the critical limitations of existing work. Our technique allows for legitimate data reads in executable memory without necessitating error-prone embedded data relocation.
- To the best of our knowledge, PXoM reveals minimal runtime overhead when compared to existing XoM tools. Our extensive evaluation demonstrates that PXoM is a viable solution for real-world adoption.

Open Source PXoM's source code and datasets are available at Zenodo to facilitate reproduction, replication, and reuse.

II. Background, Motivation, and Related Work

In this section, we provide background information on JIT-ROP attacks and the importance of addressing memory disclosure vulnerabilities. We also review existing approaches for enforcing the XoM policy on userland programs and identify their limitations, which have prompted our research. Finally, we introduce the hardware feature that we leverage to implement our fine-grained permission control mechanism and kernel-level XoM protection.

A. Overview of Just-In-Time ROP

With the advancement of fine-grained code randomization [8], [9], [13], [16], [19], traditional code-reuse attacks [42] have evolved into more sophisticated styles like JIT-ROP attacks [21], which generate ROP payloads at runtime. As illustrated in Figure 1, a typical JIT-ROP attack consists of two stages. First, attackers recursively scan code pages using memory disclosure vulnerabilities to search for gadgets (❶ in Figure 1), which typically are code sequences ending with a return instruction. In the second stage, the collected gadgets are linked together to create a payload that exploits a memory corruption vulnerability (e.g., buffer overflow or use after free) to hijack the program's control flow (❷ in Figure 1). To complete the search of the whole gadget chain within a small time window (e.g., a few seconds), JIT-ROP attackers require "*unfettered access to a large number of the code pages*" [21] to find usable gadgets quickly. Therefore, preventing disclosure of memory pages is crucial to mitigating these attacks.

Programs susceptible to JIT-ROP attacks primarily fall into the following two categories:

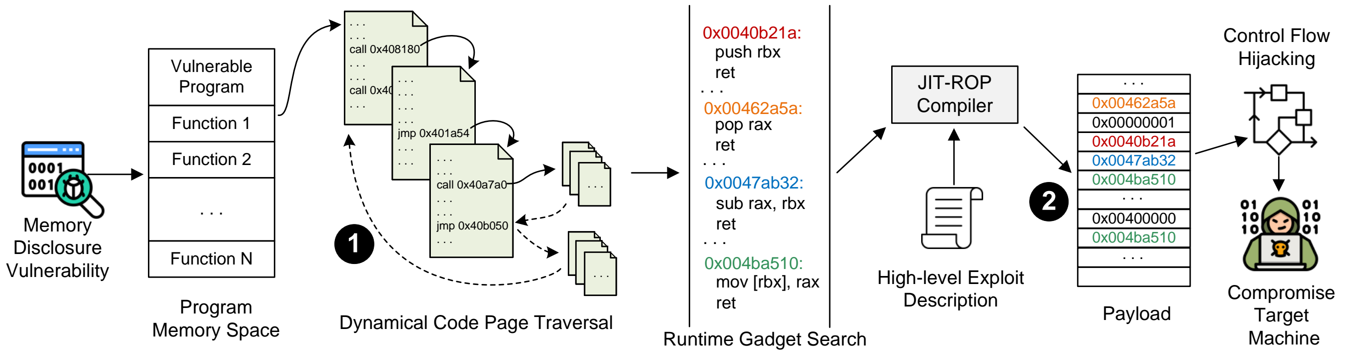

Figure 1: Overview of a typical JIT-ROP attack. Memory disclosure is the premise of a JIT-ROP attack.

Server-side programs, such as web servers and databases, which allow multiple user interactions, are particularly vulnerable to JIT-ROP attacks. An attacker can interact with the compromised program remotely, incrementally disclosing parts of its code. The search for gadgets and the construction of the malicious payload take place on the attacker's device, while the final payload is executed on the victim's machine.

Client-side programs, such as Matlab, Autodesk Maya, and JIT engines (e.g., JavaScript), are also vulnerable to JIT-ROP attacks. Attackers exploit these vulnerabilities by utilizing scripting languages. When the victim executes a malicious script, it dynamically searches for code on the victim's machine and constructs the attack payload in real-time. Among these programs, JIT engines are especially prone to exploitation. Attackers can automate exploitation by directing victims to websites hosting malicious scripts, prompting the browser to execute the exploit without user awareness.

B. Execute-only Memory Defense

The concept of Execute-only Memory (XoM) was once introduced by Multics as early as 1967 [43]. However, it was not adopted by modern operating systems and hardware until JIT-ROP emerged as an urgent threat. Next, we introduce XoM approaches designed to protect userspace software from JIT-ROP attacks.

First XoM Defense against JIT-ROP The first approach to reintroduce XoM into Linux on the x86 architecture as a defense against JIT-ROP attacks was XnR [23]. Since there was no hardware feature on x86 supporting XoM, XnR configures the PTE_PRESENT bit in PTE (Page Table Entry) of code pages as the "no present" state, causing all read operations to be intercepted by XnR's page fault handler. However, due to the substantial overhead incurred by XnR's software implementation, it makes a trade-off to allow several code pages to exist in the present state. This trade-off causes XnR to miss read operations to these co-existing code pages, leaving memory disclosure opportunities. More importantly, XnR neglected to handle legitimate read operations that point to code pages. As acknowledged by XnR's authors, XnR will be hindered by such data-in-code reads. It is clear that the two main factors limiting the deployment of XnR are hardware support and backward compatibility. These challenges have also hindered the broader adoption of XoM since its reintroduction by XnR over a decade ago.

Hardware Support When XoM was first reintroduced by XnR, XoM permissions were not supported by hardware and could only be implemented through software emulation, which resulted in significant overhead. Later, on the x86 architecture, Intel's Extended Page Tables (EPT) [35] hardware virtualization mechanism was utilized to implement execute-only permissions, improving performance to some extent. However, EPT requires programs to run within a virtual machine, and virtualization itself introduces additional performance overhead. Android previously supported XoM on the ARM architecture. However, due to implementation flaws that could lead to the failure of Privileged Access Never (PAN) [44], the flawed XoM implementation was removed beginning with Android 11 [45]. Fortunately, Intel's introduction of Memory Protection Keys (MPK) restored the ability to efficiently separate read and execute permissions, making it the preferred method for implementing execute-only permissions. The Linux kernel has begun supporting execute-only permissions at the kernel level using MPK [46]–[48]. In this paper, we also leverage MPK to efficiently enforce execute-only memory permissions. MPK allows us to overcome performance challenges of XoM, enabling us to focus on addressing the other major obstacle to the widespread adoption of XoM: backward compatibility.

Backward Compatibility A major factor limiting the widespread adoption of XoM is the challenge of protecting the vast number of precompiled programs. The XnR method [23] is built on the strong assumption that code pages do not contain any data. However, this assumption is not always valid in practice. Despite modern compilers favoring the separation of code and data, non-code bytes such as jump table data and static read-only data often appear in code sections [31], [49]. This is confirmed by Pang et al.'s SoK study on mainstream binary disassembly tools [37], which found that the mixture of code and data is very common in programs. For instance, the authors discovered 295 hard-coded bytes from the code pages of three test cases and $21,586$ jump tables embedded in the code pages of 57 programs. Inline assembly code [50] in C libraries also frequently embeds data in code sections, such as in the case of OpenSSL, BoringSSL, and FFmpeg, which use handwritten assembly to speed up their calculations. VirtualBox also employs handwritten assembly

to achieve function lazy loading and its virtual extensible firmware interface. Furthermore, if a binary file links library functions that mix code and data, its code section will also contain embedded data.

The utilization of code and data in conjunction is also required by some security solutions. One such example is KCFI [51], which places the hash value of a function's prototype in the code section via a custom LLVM pass. KCFI reads the embedded hash value to verify the control-flow integrity at runtime. As admitted by KCFI's developer, it is incompatible with execute-only memory like XnR.

Compile-time Transformation In an effort to separate data and code areas for enforcing XoM, one category of follow-up work employs compile-time transformation [24]–[26]. LR^2 [24] is an example of this approach, which compiles source code using a custom compiler and designates code and data to different memory spaces. This effectively prevents all read operations to code pages, thereby enabling XoM. However, LR^2 uses a pure software approach, which involves adding a series of stub code in front of each memory load instruction to verify the legality of read operations, leading to significant overhead. Another solution, Readactor [25], also employs a custom compiler and linker to separate code and data. It utilizes Intel EPT, a hardware-assisted virtualization technique, to manage the read permission of all code pages when mapping the virtual machine's physical address to the host's physical address. Finally, uXoM [26] provides XoM protection on `ARMv7-M` architecture for embedded devices by manipulating the Memory Protection Unit (MPU). It does this by implementing a custom LLVM pass to convert memory load instructions to unprivileged instructions.

However, all of these solutions require recompiling source code to create code-data-separated binaries, leaving pre-built legacy programs and COTS binaries unprotected. Furthermore, these approaches cannot cover handwritten assembly functions, which are often used by libraries for enhanced optimization purposes. For instance, our analysis of OpenSSL 1.1.1q's code section revealed that up to $172,058$ bytes of data are embedded in the code section.

Binary Hardening Another category of research aims to enforce the XoM policy with only binary files. HideM [27] and SECRET [29] rely on debug information (e.g., function symbols and DWARF) to identify data in code sections prior to runtime. During runtime, HideM's XoM is achieved by desynchronizing ITLB (Instruction Translation Lookaside Table) and DTLB (Data Translation Lookaside Table). This causes code and data with the same virtual address to be mapped to distinct physical addresses, effectively segregating code and data pages. HideM then redirects read operations for code pages to the separate data page. However, this revision disrupts the TLB flush mechanism, leading to performance penalties. In addition, the split-TLB feature is no longer supported—modern processors released after 2008 have replaced the split-TLB with unified second-level TLBs. NORAX [28] disassembles AArch64 stripped binaries and relocates executable data[1] to a non-code segment via binary patching. During relocation, NORAX must correctly update all references to the relocated data. Failing to do so may trigger an access violation and cause protected programs to crash. Unfortunately, updating static data references, such as those from code and the symbol table, is not a simple task [52]. Even more challenging is updating references generated dynamically, such as those from the global offset table (.got) and read-only global data (.data.rel.ro) [6]. Furthermore, our findings reveal that NORAX's embedded data identification strategy may fail to properly handle cases where code is misidentified as data. In the event of such an occurrence, NORAX's functionality will cease to operate properly. For instance, if a small function is mistakenly classified as embedded data, the references to the function (e.g., through a function pointer) are also updated to a non-executable area, which may cause the protected program to crash when the function pointer is dereferenced at runtime.

Destructive Code Reads To address the issue of XoM methods not supporting legitimate data reads within code pages, Heisenbyte [31] proposed a variant of XoM mechanism, called Destructive Code Reads (DCR). Heisenbyte allows memory disclosure but prevents executing the previously disclosed code by destroying the disclosed code right after it is read. Heisenbyte marks each executable memory page as execute-only and maintains a duplicate copy for each execute-only page. When a read operation occurs on the execute-only page, Heisenbyte overwrites the read data with random bytes and returns the corresponding data values from the duplicate page. Thus, legitimate read operations for data-in-code work correctly, but attackers cannot run disclosed executable memory. NEAR [32] is another DCR approach building upon Heisenbyte, providing a more reliable and efficient memory destruction mechanism. Although DCR successfully supports legitimate data reads within code pages, the code inference attacks proposed by Snow et al. [33] have completely undermined DCR's security guarantees. The core idea of code inference attacks is to disclose a piece of code but not to execute it. Instead, another piece of code that is strongly related to it will be executed, such as an exact same copy of the disclosed code in a different memory area, or the relevant code that can be predicted based on the disclosed ones. Despite the possible evasion to DCR, it still provides valuable insights for advancing XoM. It underscores the critical challenge of preventing code exposure while simultaneously permitting legitimate reads to embedded data. This inherent dilemma serves as a compelling motivation for our current research.

Comparison of XoM Techniques Table 1 presents a comparison of various XoM approaches that aim to provide userland software protection. XnR does not require source code or binary rewriting. However, it doesn't support legitimate reading of embedded data because it assumes no presence of data residing in executable code areas. Furthermore, the N-page window of XnR leaves an attack surface for adversaries.

[1] NORAX refers to data residing in executable code regions as "executable data," while we refer to executable data as "embedded data" in the following sections.

Table 1: Comparison of representative XoM approaches that protect userland programs.

	No Source Code Needed?	No Debug Symbols Needed?	Support Data-in-Code Reads?	Hardware[1] Feature	No Binary Patching Needed?	Architecture	Runtime Slowdown	Memory Overhead
XnR [23]	✓	✓		N/A	✓	x86/x86-64	8.4%	Negligible
LR^2 [24]		N/A		N/A	N/A	ARMv8	6.6%	Negligible
Readactor [25]		N/A		EPT	N/A	x86/x86-64	2.8%	Negligible
uXoM [26]		N/A		N/A	N/A	ARMv7-M	7.3%	High
HideM [27]	✓		✓	Split-TLB[2]	✓	x86-64	1.4%	Small
NORAX [28]	✓	✓		AP/XN Bits		ARMv8	1.2%	Small
SECRET [29]	✓			N/A		x86	14.4%	High
Heisenbyte [31]	✓	✓	✓	EPT		x86-64	18.3%	High
NEAR [32]	✓	✓	✓	EPT		x86-64/ARMv8	5.7%	Small
PXoM	✓	✓	✓	MPK	✓	x86-64	0.36%	Negligible

[1]In this column, EPT, TLB, AP/XN, and MPK represents Extended Page Table, Translation Lookup Table, Access Permission, eXecute Never, and Memory Protection Keys, respectively. "N/A" means XoM is achieved using page table manipulation [23] or a form of software-fault isolation [24], [26], [29].
[2]The split TLB technique is not supported anymore by modern x86 processors since the Nehalem microarchitecture (released in 2008).

Compile-time transformations require the presence of source code, which fails to protect pre-compiled legacy applications. Binary hardening methods, on the other hand, can work with binaries, but only HideM supports the reading of embedded data. However, HideM relies on an obsolete hardware feature and changes the normal cache model, making it less compatible. NORAX's binary patching may fail to update data references, which changes the original functionality of the protected program. DCR methods can work on binaries and support the reading of embedded data, offering the best compatibility among all previous methods. Unfortunately, their protections can be bypassed by code inference attacks [33]. Additionally, XoM implementations via software emulation, such as LR^2, uXoM, and SECRET, incur relatively high overhead. In conclusion, these limitations highlight the need for further research in developing a practical XoM technique.

In contrast, as demonstrated in §VI and §VII, PXoM effectively thwarts the disclosure of executable memory while incurring minimal performance and memory overhead. Besides, PXoM does not require source code or debug information. At last, PXoM does not interfere with the original operating system or architectural mechanisms, and unprotected programs remain unaffected by PXoM's kernel components.

C. Memory Protection Keys

Intel Memory Protection Keys (MPK) is a hardware feature that enables stricter permission control on code pages without the need for page table modifications. The MPK mechanism uses a Protection Key Rights Register (PKRU) to maintain access rights of individual keys associated with specific pages. It supports three different page permissions: read & write, read-only, and no access. Notably, MPK controls read and write permission on memory pages, while traditional permission management mechanisms continue to manage execution permission. The MPK mechanism can be utilized to configure a memory page's permission as execute-only by disabling the page's read and write permissions. One of the significant advantages of MPK is its high performance. The processors only need to execute a non-privileged instruction (i.e., WRPKRU) to update PKRU, which takes less than 20 cycles and does not require any TLB flush or context switching [53]. However, as MPK keys are localized to each thread, it may lead to inconsistencies between MPK keys of different threads within the same process. To ensure the synchronization of execute-only MPK keys between different threads within a process, we have utilized the synchronization primitive provided by libmpk [36].

D. Kernel-level XoM Protection

PXoM and related works [23]–[29] consider the kernel to be part of the Trusted Computing Base and thus do not protect against kernel memory disclosure. Another parallel direction is kernel-level XoM protection, as the kernel itself may be exploited under certain circumstances. For example, ret2usr attacks [54]–[57] can redirect control and data flow to user space, compromising the entire system. KHide [58] and kR^X [59] counteract kernel-level JIT-ROP attacks by enabling XoM protection for kernel memory. They both rearrange the memory layout of the kernel space, placing executable code in execute-only areas and readable data in read-only areas. KHide [58] employs the hardware feature Hardware Assisted Paging (HAP) to enforce the XoM policy by mediating access on HAP violation, while kR^X [59] utilizes Intel Memory Protection Extension (MPX) to enforce XoM permission in a more efficient manner. However, Intel has discontinued MPX support since the 10th generation of Intel Core processors in 2019 [60]. IskiOS [61] simply uses MPK to revoke the read permission of kernel's code pages. However, their solution is not applicable to stripped binaries as it does not address the issue of legitimate embedded data reads.

E. Control-Flow Integrity

A precise implementation of Control-Flow Integrity (CFI) offers significant potential to safeguard applications against ROP attacks by preventing control-flow hijacking. Currently, hardware mechanisms such as ARM's Pointer Authentication Code (PAC) [62] and Intel's Control-flow Enforcement Technology (CET) [63] provide support for CFI enforcement. Although CFI can still potentially be bypassed under specific circumstances [64]–[69] and may introduce performance overhead [70], it can be deployed alongside other defense mechanisms, thus providing an additional layer of security

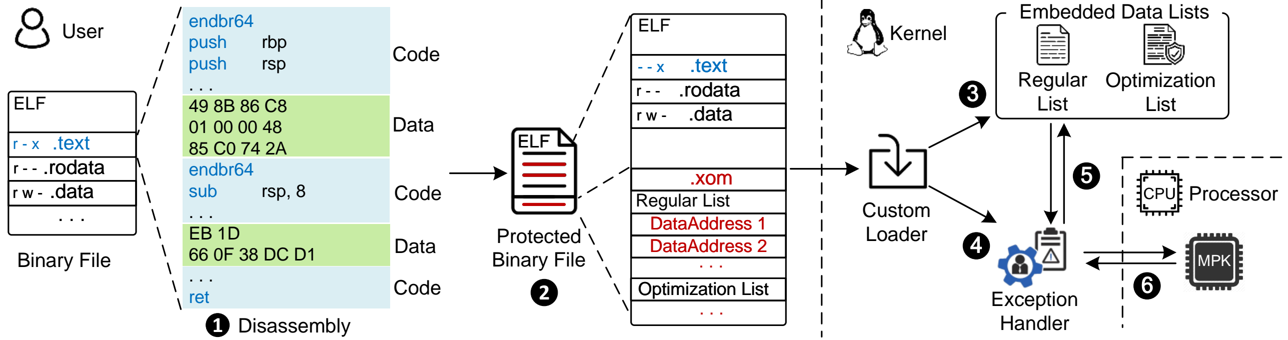

Figure 2: Overview of PXoM.

protection. From a defense-in-depth [71] standpoint, it is imperative that a critical system incorporates multiple complementary security defenses in practice.

III. Threat Model

PXoM aims to defend against JIT-ROP by preventing attackers from dynamically disclosing memory, based on a well-defined adversary model. The model includes the following assumptions:

- W⊕X: The target system ensures that the executable and writable permissions cannot coexist on the same memory page. This assumption forms the basis of ROP defenses. Otherwise, attackers could simply execute the injected shellcode directly, without the need for ROP techniques.
- Randomization: The target program uses a fine-grained code randomization technique, which frustrates adversaries to determine the protected program's memory layout in advance.
- Control-Flow Hijacking: The target program is vulnerable to memory corruption attacks that allow the adversary to hijack the control flow.
- Transparent Configuration: The adversary has knowledge of the target system's configuration, as well as access to the source code of the target program.

This adversary model is consistent with previous offensive and defensive papers [21], [23], [25], [27], and specifically aligns with the robust model introduced in JIT-ROP attacks [21]. We exclude side channels and self-modifying binaries protection from our threat model, because they are outside the scope of this paper and are also excluded by other peer works.

Crane et al. [25] pointed out that there still exists an *indirect memory disclosure attack* that can infer the code layout without directly reading the code pages by harvesting code pointers in stack and heap. They proposed a method to prevent indirect memory disclosure by redirecting the code pointers to an unreadable trampoline, and thus solved the indirect memory disclosure problem. As this defense still requires XoM protection to ensure its effectiveness, we focus on addressing the remaining issues in XoM protection. PXoM aligns with the constraint acknowledged by NORAX [28], which also works on COTS binaries.

IV. Overview

Our study continues the line of research on retrofitting XoM into stripped binaries. One of our design goals is to avoid relocating *embedded data* via binary patching. To this end, we develop a new fine-grained memory permission control mechanism, enabling the accommodation of legitimate data-in-code reads. Figure 2 shows PXoM's architecture that bridges all layers of the software stack.

User-space Components To determine the areas that are authorized to read, we first identify all embedded data in the binary prior to runtime. To circumvent the inherent complexity of precisely identifying embedded data, we employ a Unidirectional Disassembly strategy (❶ in Figure 2). This disassembly strategy ensures no embedded data will be identified as code. Subsequently, we append the list of embedded data to the end of the protected binary file and revoke the code segment's read permission (❷). In scenarios where data-in-code reads occur frequently, we also customize an optimization policy to speed up the read-legality check. We create an independent *optimization list* to store the embedded data that are frequently accessed (the right side of ❷). Please note that in this step, we do not rewrite the binary code. Instead, we simply add the addresses of embedded data to the end of the binary while marking the code segments as execute-only.

Kernel Components At the kernel level, we modify the binary loader in Linux kernel to load the protected binary. In addition to loading the protected binary, the modified binary loader also loads two embedded data lists into kernel-space memory (❸ in Figure 2) and initializes the exception handler (❹). When mapping the code segment, the custom loader loads it as execute-only using the MPK mechanism. The exception handler is responsible for ensuring the legality of data-in-code reads based on two embedded data lists. It also dynamically adjusts the optimization list on-the-fly (❺). If the address of a read request lies in either the regular list or optimization list, the exception handler will allow this read request. Otherwise, the read request will be rejected, and PXoM will terminate the process and save the context information for further forensics investigation. The exception handler uses the MPK mechanism to efficiently check read request legality (❻), resulting in very low runtime overhead.

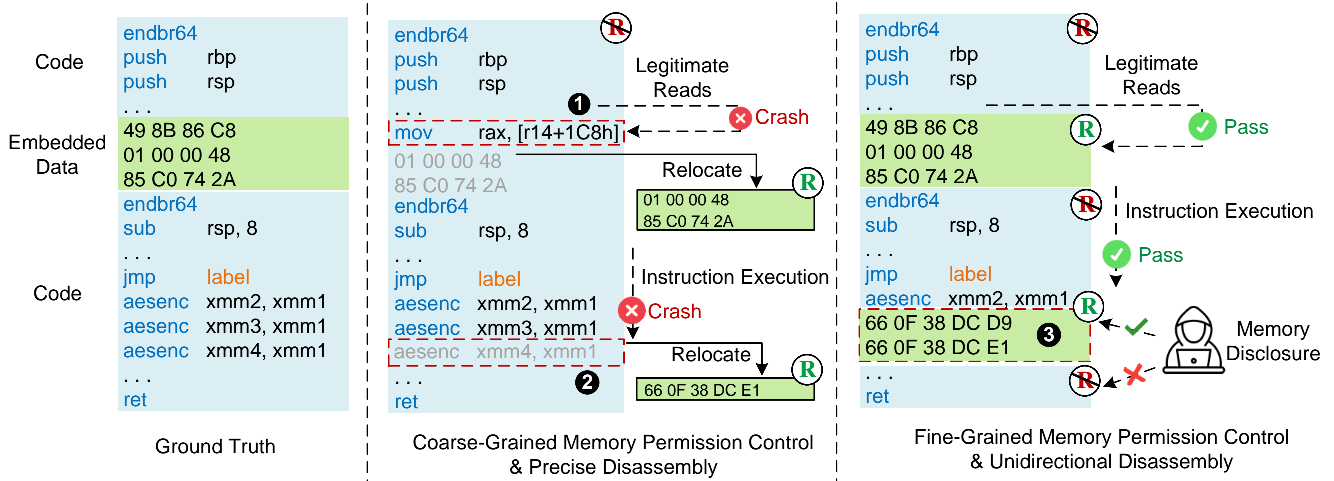

Figure 3: The left side shows a code page containing embedded data. The middle section illustrates the previous XoM based on core-grained memory permission control with precise disassembly. The right side depicts the effect of PXoM via fine-grained memory permission control with Unidirectional Disassembly.

V. Design

In this section, we follow the workflow of hardening an application to describe each component of PXoM.

A. Fine-Grained Memory Permission Control

The management of permissions in modern OSs is limited to memory pages, which we call coarse-grained control over memory permissions. As a result, previous XoM methods have to relocate embedded data within code pages and update all references to ensure that the program can access them. However, the precise identification of code and data within binary remains an undecidable problem [52]. Previous disassembly efforts [72]–[77] aimed to minimize both code-to-data and data-to-code misidentifications, which we can refer to as *Precise Disassembly*. For example, the left side of Figure 3 shows a code page containing an embedded data block, while the middle section displays the Precise Disassembly result of this code page. In previous XoM methods, erroneous identification of code as data (❷ in Figure 3) leads to the inadvertent relocation of code outside code pages, thereby altering program semantics. On the other hand, when embedded data are misinterpreted as code, the legitimate read of this data will be prohibited (❶ in Figure 3). Both of these errors pose significant crash risks. Moreover, updating references to relocated embedded code presents a substantial challenge. Failure to update references to embedded data following relocation may result in program crashes when attempting to read these segments.

We have implemented a fine-grained memory permission control mechanism to assign different permissions to various memory regions within the same memory page. This mechanism enables the removal of read permissions for code segments while retaining read permissions specifically for embedded data within the same memory page. This approach serves the dual purpose of safeguarding code against disclosure while facilitating legitimate reads of embedded data. We capture all read requests in executable areas and scrutinize their legitimacy in the kernel's page fault exception handler, which we will detail in §V-E. However, as previously noted, employing Precise Disassembly may result in both code-to-data and data-to-code misidentifications. In the event of code-to-data misidentification, although it may potentially expose small code segments to the risk of memory disclosure, the program can still function correctly because PXoM does not relocate embedded data. Conversely, misinterpreting any data as code may lead to the program crash caused by legitimate read attempts. Therefore, we require a disassembly strategy to circumvent the inherent complexity of precisely identifying embedded data, thereby preventing potential crashes.

B. Unidirectional Disassembly Strategy

We employ a disassembly strategy designed to avoid misidentifying data as code, while tolerating some code being misidentified as data, in order to meet the requirements of our fine-grained memory permission control mechanism. Rather than attempting to precisely identify all embedded data, our strategy identifies a superset of embedded data that includes both all actual embedded data and a very small amount of code. We refer to this approach as Unidirectional Disassembly. We use Figure 4(A), a code section containing embedded data, as an example to demonstrate step-by-step process of the Unidirectional Disassembly.

Initially, the entire code section is marked as embedded data superset (as shwon in Figure 4(B)). Then, we apply the recursive traversal algorithm [78], following the control flow from the program entry point to identify the code located on the main paths. Recursive traversal disassembly partially meets the requirements of our fine-grained memory permission control mechanism by avoiding any data-to-code misinterpretation. This method performs disassembly exclusively on instructions, tracking the program's control flow and thereby preventing the misidentification of data as code. We then exclude the

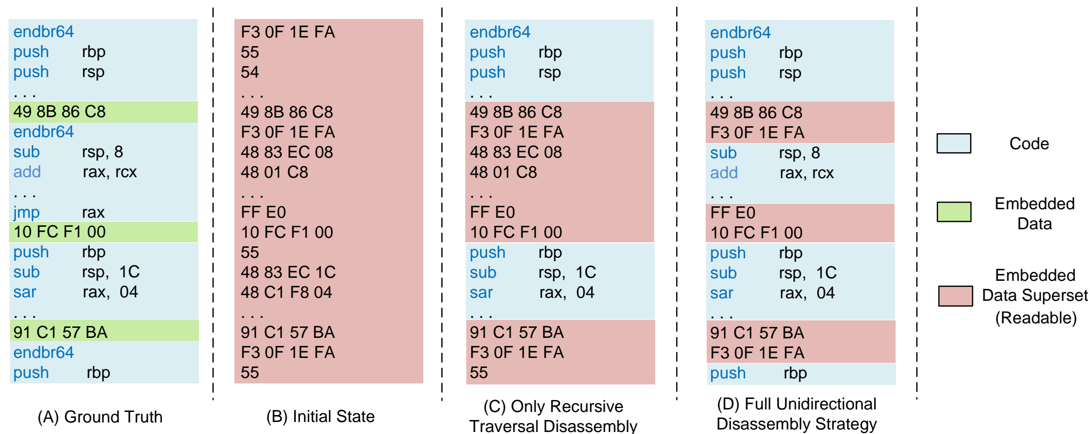

Figure 4: Workflow of Unidirectional Disassemblystrategy. The blue, green, and red sections represent code, embedded data, and the superset of embedded data, respectively. The left side shows a code section containing embedded data. The middle section illustrates the embedded data superset when only recursive traversal disassembly is conducted. The right side demonstrates the minimized embedded data superset after applying the full Unidirectional Disassembly strategy.

identified code from the superset, resulting in a smaller superset (Figure 4(C)). However, recursive traversal disassembly struggles with handling indirect calls and unreachable functions [78], potentially missing up to **49.35%** of the code on average [37]. This can lead to a large embedded data superset, thereby exposing too many readable areas to adversaries. To further reduce the superset, we conduct multiple additional analyses to uncover missed code entry points, subsequently applying recursive traversal disassembly to these identified entry points. During disassembly from each entry point, the identified code is excluded from the superset, thereby minimizing the embedded data superset (Figure 4(D)). Specifically, our analyses include examining jump tables, frame unwind information, address-taken functions [6], and employing function entry identification heuristics [79] to identify additional code entry points that were not reached by recursive traversal disassembly. We also provide a detailed algorithm for the Unidirectional Disassembly, please see Appendix C.

After obtaining the embedded data superset, we make the entire superset readable and prohibit read permissions for all remaining executable areas using our fine-grained memory permission control mechanism. This ensures that all legitimate embedded data reads are confined to this superset, while any code disclosure attempts outside of this superset are prohibited. For example, as shown in the right section of Figure 3, no embedded data are misinterpreted as code, but two instructions are misidentified as embedded data (❸ in Figure 3). As a result, in addition to real embedded data, a small amount of code also retains the read permission. For the sake of convenience, we will refer to the embedded data superset simply as embedded data in the following context, as the entire superset will be made readable, and the misidentification of code as data does not affect the executability of the code. In §VI, we will measure the coverage of disassembly results and evaluate whether embedded data are exploitable. The results demonstrate that our approach provides sufficient protection against memory disclosure without compromising practicality.

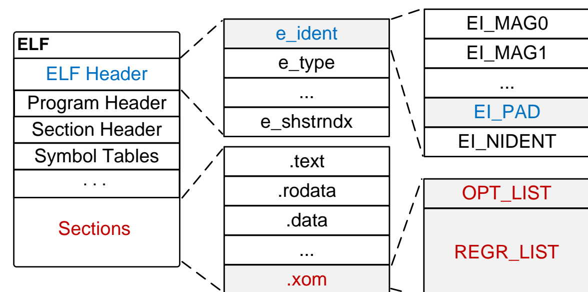

Figure 5: New ELF format for PXoM protected binary file.

C. New ELF Format

We define a new ELF format to interact with PXoM's kernel components. Our minor changes to the original ELF format include utilizing the reserved field and optional section of the ELF file format to store PXoM flags and the embedded data list. Figure 5 shows a visual representation of the new ELF file format we have devised. First, we use a reserved byte in the ELF header as the PXoM specific flag byte, **XOM_ENABLE**, to indicate whether the program is protected by PXoM. This byte is the second byte in the EI_PAD array, which is a field of e_ident in the ELF header. By checking this byte, our custom loader can decide whether to enable PXoM protection. Afterward, the embedded data list is included in an optional section called *.xom*. The OPT_LIST and REGR_LIST represent the optimization list and regular list, respectively.

Please note that the new ELF format remains backward compatible with non-customized loaders because they will disregard the XOM_ENABLE flag and the .xom section. The unaltered kernel can execute PXoM-protected binaries without any issues as conventional programs.

D. Custom Loader

The custom loader is a kernel component of PXoM that loads protected binaries and initializes related structures in the kernel. To determine if PXoM's protection is enabled, the

loader checks the XOM_ENABLE flag in the ELF header. If yes, the loader loads the embedded data list stored in the .xom section (❸ in Figure 2) and initializes an exception handler to ensure data-in-code reads are legitimate (❹ in Figure 2). The exception handler is also a kernel component to check the legitimacy of data-in-code reads, and we will introduce it later. If the PXoM flag is not enabled, the standard binary file-loading process takes over. To prevent attackers from disclosing or tampering with PXoM information, all PXoM-related metadata is stored in kernel memory. The PXoM flag, optimization list pointer, and regular list pointer are stored in the `task_struct`. Each process has its own task_struct object, which stores the context of the process. Once the lists have been loaded, the custom loader begins mapping the code segment into memory. During this mapping process, the loader assigns an execute-only PKey, and sets the code section as execute-only with this PKey. The PKey is a part of the MPK mechanism to set the permission for a group of pages. This allows our exception handler component to detect any read operation to code pages and determine if it is a legitimate data-in-code read or a malicious memory disclosure attempt.

E. Exception Handler

We implement an exception handler based on the original page fault handler for the MPK mechanism to prevent memory disclosure. With the read permission of all code pages removed via the MPK mechanism, any read request to a code page will trigger a page fault and be caught by our exception handler. The exception handler then determines if the target address is located in the embedded data areas. If not, we promptly determine that the running program is under a memory disclosure attack and terminate the compromised process, while saving the context information for further forensics investigation. Please note that legitimate read operations for data embedded in the code will not trigger PXoM's attack response. Instead, we take the following actions to allow such a data-in-code read: 1) we restore the read privilege of the target page to allow it to be read temporarily. 2) We set the single-step trap flag to execute only the read instruction and halt at the next instruction. 3) Once the legal read operation is complete, we revoke the code page's read permission again and clear the single-step trap flag to resume the program's normal execution. To keep track of whether a read operation is legal, we maintain a flag, called **XOM_ALLOW_READ**, in the `task_struct` and set it to false by default. Once a read operation occurs and is determined to be legal, we set the XOM_ALLOW_READ to true. The single-step trap handler uses this flag to determine whether to allow the read operation. If true, the data read operation is permitted. Subsequently, upon completion of the read operation and restoration of the page permission, the XOM_ALLOW_READ flag is reset to false.

Next, we define the legitimacy of data-in-code reads. A legitimate data-in-code read should not access memory out of the embedded data list. If a data-in-code read only includes the whole or a subsection of a data area that is in the embedded data list, we take it as a legitimate read. On the other hand,

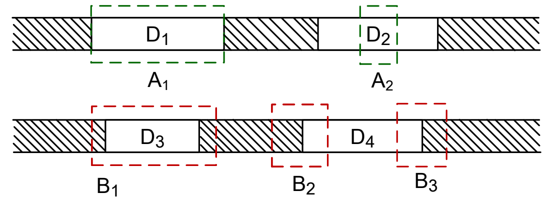

Figure 6: Different types of data-in-code read requests. A read request may access multiple bytes at the same time.

if a data-in-code read covers a part of memory that is absent from the embedded data list, it is deemed an illegitimate read. Figure 6 illustrates all legitimate and illegitimate reads. The $D_1 \sim D_4$ represent embedded data areas recorded by the embedded data list, while the shadowed areas represent code areas. The data-in-code reads A_1 and A_2 in the first line are legitimate data reads, while B_1, B_2, and B_3 in the second line are illegitimate data reads. On the x86-64 architecture, it is possible for a data-in-code read to access multiple bytes using a single instruction. For instance, the `MOV` instruction enables the retrieval of up to 8 bytes of data, while Streaming SIMD Extensions allow simultaneous access to a maximum of 16 bytes of data. A_1 covers the entire data block of D_1, while A_2 covers a portion of D_2. These areas fall within the boundaries of D_1 and D_2, thereby qualifying as legitimate reads. Conversely, the regions of B_1, B_2, and B_3 intersect portions of code segments, and therefore, they are regarded as memory disclosure attempts.

In the scrutiny of a read request, PXoM faces potential performance bottlenecks when iteratively navigating an extensive list of embedded data, a concern exacerbated in programs featuring a large number of code-data interleaving cases. Our empirical observations of real-world programs reveal that, in scenarios where data-in-code reads occur frequently, the read targets tend to cluster around a confined subset of embedded data. For example, during the execution of AES-256-CBC encryption in OpenSSL, the program exclusively accesses a mere 15 out of 8142 embedded data blocks. This motivates us to implement a cache-like optimization policy that accelerates the legitimacy determination of frequently-read embedded data. We segregate the high-frequency read embedded data into a separate *optimization list*, while preserving other embedded data in a *regular list*. We ensure that the optimization list remains concise, prioritizing its iteration when validating the legitimacy of data-in-code reads. Our strategy for creating the optimization list encompasses both static and dynamic policies. Under the static policy, we consider the frequency of references to embedded data, relegating embedded data with over 10 references to the optimization list. Meanwhile, the dynamic policy involves real-time monitoring within the exception handler, recording the frequency of reads for embedded data. Data surpassing 100 reads dynamically qualifies for inclusion in the optimization list. We have empirically determined the threshold of 10 references and 100 reads to achieve the optimal performance. PXoM turns on this optimization policy by default. For the evaluation of how this

optimization policy affects the performance of high-frequency inline data reads, please refer to Appendix C.

VI. Security Evaluation

In this section, we first examine the outcomes of Unidirectional Disassembly to gauge the completeness of PXoM's protection. Following this assessment, we delve into an exploration of PXoM's attack surface to ascertain its effectiveness. Through a series of experiments, our findings consistently demonstrate that PXoM offers a comprehensive defense mechanism against JIT-ROP attacks. Please be aware that the "embedded data" in this section is actually the superset of embedded data, as described in §V-B.

A. Unidirectional Disassembly Result Analysis

Our proposed Unidirectional Disassembly strategy ensures the comprehensive coverage of data within code areas, which endeavors to maximize the identification of code areas while ensuring *zero* misinterpretation of embedded data. We present several metrics in this section to gauge the correctness and extent of coverage achieved by this methodology.

The first metric is the Code Coverage, which represents the ratio of disassembled results to the total code bytes. Code coverage is calculated by Equation 1 (CC is short for "Code Coverage."):

$$CC = \frac{Disassembled\ Code\ Bytes}{Real\ Code\ Bytes} \tag{1}$$

This metric illuminates PXoM's efficacy in safeguarding the actual code present in binary files. The core of this capability is rooted in PXoM's ability to restrict read access solely to the code sections identified through the disassembly process, while relaxing read access for the remaining regions. While the real code is more likely to be used as gadgets by attackers, the embedded data could also be used as gadgets under certain conditions. Hence, we introduce the second metric, the Overall Coverage, which denotes the proportion of disassembled code relative to all executable bytes, comprising real code and embedded data. The Overall Coverage is computed by Equation 2 (OC is short for "Overall Coverage."):

$$OC = \frac{Disassembled\ Code\ Bytes}{Real\ Code + Embedded\ Data} \tag{2}$$

The third metric is the Number of Embedded Data Blocks, representing how many embedded data blocks in the binaries. The last metric is the Average Embedded Data Block Size. These two metrics indicate the distribution density of embedded data blocks within the program.

To gauge the correctness and extent of coverage achieved by our Unidirectional Disassembly, we use extensive of datasets, including open source applications and COTS closed-source applications, to evaluate the above four metrics. For open source applications, we can extract their ground truth, allowing us to accurately measure these metrics. For the COTS closed-source applications, although we are unable to measure their code coverage (due to the unavailability of their ground truth), we still evaluated their overall coverage, number of embedded data blocks, and average embedded data block size. This is still meaningful in providing supplementary evidence of PXoM's protection capabilities.

Table 2: Disassembly result of Unidirectional Disassembly on open source applications. The "EDB" in the fourth and fifth columns represents "Embedded Data Block."

Benchmark	Code Coverage	Overall Coverage	#. of EDB	Avg. EDB Size (B)
SPEC 2017	97.58%	96.34%	3020	30
Webservers	99.39%	98.96%	593	9
Databases	97.67%	98.29%	9408	17
OpenSSL	95.61%	86.43%	8142	31
Pang et al. [38]	96.01%	95.79%	1096	52
Overall	97.07%	95.29%	4290	31

Analysis of Results on Open Source Applications We evaluate a wide variety of stripped binaries compiled with different optimization options, encompassing SPEC CPU 2017 benchmarks, web servers such as Nginx, Apache, and Lighttpd, along with databases including MySQL, MongoDB, Redis, and SQLite. We also evaluate a substantial binary dataset obtained from the recent binary disassembly study by Pang et al. [38].This dataset consists of approximately $4,000$ binary files with a total size of around 20GB, serving as a reliable source of ground truth for disassembly assessments. Additionally, they released a compilation toolchain capable of extracting ground truth from compiled binaries. We utilized this toolchain to compile the open source applications and extract ground truth.

The evaluation results of above four metrics for open source applications is shown in Table 2. The lowest code coverage, obtained in OpenSSL at 95.61%, can be attributed to the extensive use of handwritten assembly. The average code coverage stands at 97.07%, implying that PXoM can protect a significant portion of the code present within the binary. For the overall coverage, OpenSSL still reveals the lowest OC metric at 86.43%. However, the average overall coverage of 95.29% suggests that PXoM can safeguard the majority of executable memory from potential attackers. The fourth and fifth columns of Table 2 provide insights into the number of embedded data blocks and their average size in bytes. The latter indicates the amount of consecutive bytes that attackers can potentially disclose if they manage to identify readable embedded data blocks. The average size of an embedded data block is a mere 31 bytes, which implies that attackers can consecutively disclose only 31 bytes on average when they locate a readable block in the executable area.

Analysis of Results on COTS Closed-Source Applications We collected 15 different COTS closed-source applications and analyzed them using Unidirectional Disassembly. Due to the lack of ground truth information for these closed-source binaries, we are unable to definitively identify the actual code and embedded data, preventing us from reporting a precise Code Coverage metric. However, we assessed other relevant metrics, including Overall Coverage, the number of embedded data blocks, and the average size of these blocks. The results are shown in Table 3. The first column of Table 3 lists the

application names and their respective versions used in our evaluation. The second column presents the Overall Coverage. The lowest Overall Coverage was observed in OracleDB, at 86.44%. On average, the Overall Coverage is 96.94%, demonstrating that PXoM can protect approximately 96.94% of the code in these COTS applications from being exposed to attackers. The third column displays the number of embedded data blocks, with an average of 1,217 across the analyzed applications. The last column shows the average size of the embedded data blocks, which is 55 bytes. This means that if attackers gain access to an embedded data block with read permissions, they would only be able to read an average of 55 bytes.

Case Studies Since we cannot obtain the ground truth for COTS closed-source applications, we manually verified some of embedded data and used them as case studies to illustrate how these COTS applications utilize embedded data. For detailed case studies, please refer to Appendix B.

Our experimental results are encouraging, as they validate our claim for embedded data identification. These results provide further assurance that PXoM can effectively impose execute-only permission on code areas. Next, we will present additional evidence to support that the residual embedded data are insufficient to construct a payload.

B. Attack Surface Analysis

To tolerate legitimate data-in-code reads, we allow embedded data to remain readable. Nonetheless, the disassembly process of PXoM may still experience some false positives, whereby code is misidentified as data. Consequently, the embedded data list provided by PXoM includes both the true embedded data and some misidentified code, as shown in the second and third columns of Table 2. Given the embedded data list delivered by PXoM, it is imperative to evaluate adversaries' capabilities to harvest reusable gadgets and subsequently construct an attack payload. We conduct a gadget search experiment on the embedded data sections for each binary in the dataset utilized in Section VI-A. The objective of this experiment is to quantify the number of gadgets that can be identified within embedded data regions.

ROP Gadget Search After applying the ROP gadget search tool ROPGadget [80], we found that available gadgets are a rare commodity, even for binaries that contain a significant amount of embedded data. On average, only *seven* gadgets can be extracted from embedded data, which are comprised of 287 small blocks. That means these gadgets are distributed among 287 different locations throughout the entire code section. These seven gadgets represent the upper limit of potential adversary exploitation. With fine-grained randomization enabled, adversaries lack prior knowledge of where the data blocks are distributed, making it extremely difficult to disclose all the gadgets at once.

As embedded data consist of small data blocks distributed in the code section, we present the average embedded data block size in the last column of Table 2. For the overall dataset, the average embedded data block size is only 31 bytes, and the total of embedded data accounts for 4.71% (i.e., 1-95.29%) of the whole code section. This indicates that if an adversary were to choose an address randomly in the code segment and attempt to disclose code, the probability of this address landing in the readable area is only 4.71%. If attackers are fortunate enough to find an embedded data block that can be read through this 4.71% probability, then the average amount of data they can disclose is only 31 bytes. This is insufficient to build a ROP chain, as previous ROP gadget search papers have supported [81], [82]. The "microgadgets" technique [82] attempts to use the gadgets restricted to 2 or 3 bytes in length to construct the ROP chain; however, it needs to scan at least 3MB of code to find enough microgadgets. Schwartz et al. [81] developed an offline verification technique to facilitate ROP attacks necessitating Turing-completeness, but it still requires at least 20KB of code to construct a complete payload chain.

Table 3: Disassembly result of Unidirectional Disassembly on COTS closed-source applications. The "EDB" in the fourth and fifth columns represents "Embedded Data Block." The "VMWare" represents VMWare Workstation Pro.

COTS Application	Overall Coverage	#. of EDB	Avg. EDB Size (B)
Skype (8.129.0.202)	99.98%	718	58
DaVinci Resolve (19.0.1)	98.39%	425	41
IBM DB2 (15.5.9)	98.95%	401	15
LiteSpeed (6.3.1)	99.91%	43	68
Matlab (R2024b)	98.94%	690	125
AutoDesk Maya (2025_2)	98.67%	1254	17
OracleDB (193000)	86.44%	577	31
Spotify (1.2.45.454)	99.75%	1637	39
Intel DPC++ (2.1.79)	92.59%	2233	68
Intel Fortran (2.1.80)	92.19%	2559	75
Steam (1726604483)	99.27%	1190	16
TeamViewer (15.58.4)	94.49%	2461	25
Unity (6000.0.20f1)	98.90%	434	51
VMWare (17.6.0)	98.65%	780	109
Zoom (6.2.0)	97.00%	2740	90
Overall	96.94%	1217	55

Case Studies To show the effectiveness of PXoM protection on real-world threats, we leverage the JIT-ROP attack framework, jitrop-native [83], to exploit a Nginx arbitrary memory disclosure vulnerability. We also conducted an experiment to show that the WRPKRU instruction does not pose a threat to PXoM protection. Please refer to Appendix A for details.

Our comprehensive experimental results demonstrate that PXoM can effectively safeguard programs against the threat posed by JIT-ROP attacks.

VII. Performance Evaluation

Our performance experiments evaluate PXoM from five aspects: 1) performance on microbenchmarks; 2) performance on macrobenchmarks; 3) investigation into overhead causation; 4) performance on programs exhibiting high-frequency embedded data reads and the impact of our optimization; 5) performance comparison with existing work.

Our evaluations were conducted on a desktop machine featuring Intel Core i9-13900KF and 64GB of memory, running

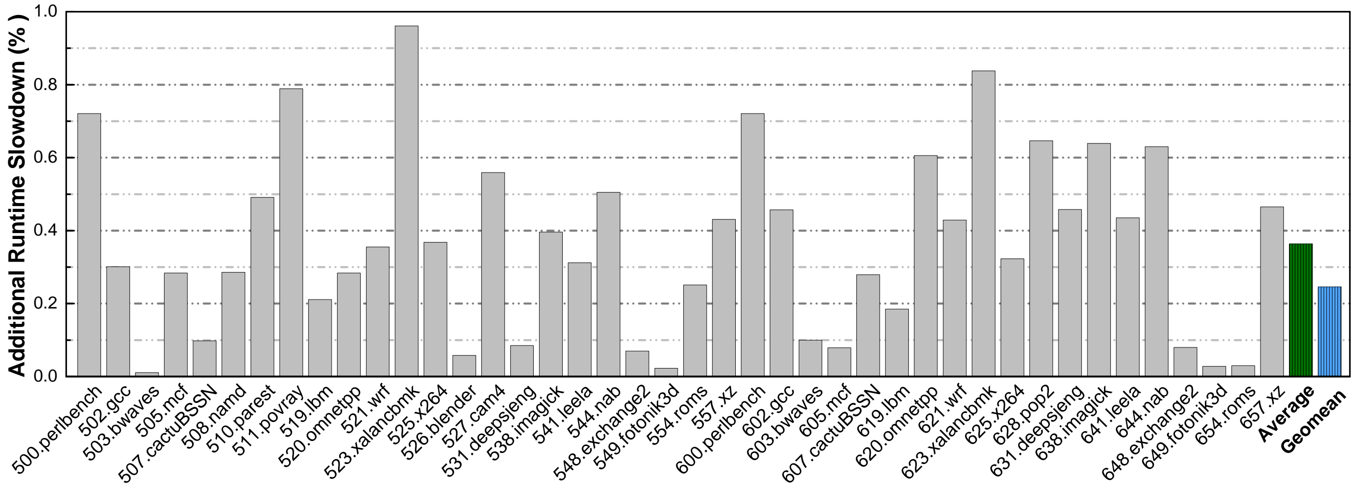

Figure 7: Additional runtime slowdown (%) of PXoM on SPEC CPU 2017 (*ref* workload).

Table 4: Time for kernel operations related to process creation and page fault handling (in μs). Smaller is better. In the first row are the names of benchmarks in lmbench. The "Prot Fault*" in the last column shows the overhead when data-in-code read-legality check is triggered.

Kernel	Fork Proc	Exec Proc	Page Fault	Prot Fault	Prot Fault*
Standard	109	339	0.776	0.484	0.484
PXoM	110	341	0.780	0.487	1.548
Overhead	0.92%	0.60%	0.52%	0.62%	3.20X

Ubuntu 23.10 with Linux kernel 6.5.0. Our evaluation results indicate that PXoM's protection results in negligible additional overhead, ranging from 0.24% to 0.82% on average. Even in OpenSSL, which contains a substantial amount of data-in-code reads, the overhead caused by PXoM is only 0.82%. The memory overhead incurred by PXoM is also minimal, with an average of only 0.13%. The following subsections focus on the measurement of runtime slowdown, whereby we ran both the standard version and the PXoM-protected version for each binary, respectively.

A. Microbenchmarks

Compared to the standard Linux kernel, we made modifications to the kernel's binary loader, page fault exception handler, and process context structure. Therefore, we run lmbench [39] on both the standard Linux kernel and the modified Linux kernel to assess the performance overhead induced by our kernel modifications.

Table 4 shows the running time for kernel operations related to process creation and page fault handling. The *Fork Proc* and *Exec Proc* are process creation operations using *fork* and *exec*. They resulted in an overhead of 0.92% and 0.60%, respectively, due to the additional steps required for loading XoM metadata. The *Page Fault* shows the overall page fault handling overhead, with a 0.52% overhead. The last two *Prot Fault* show the protection fault handling overhead without and with triggering the data-in-code read-legality check. The first Prot Fault is the overhead on regular prot fault process, without triggering the read-legality check. However, the handler still needs to check if the PXoM is enbaled, resulting a 0.62% overhead. In the second *Prot Fault**, we deliberately trigger the data-in-code read legality checks to evaluate the performance overhead on legality checking process. Unlike the first four configurations that incur negligible overhead, this configuration incurs a notable overhead with a 3.20 times slowdown. However, this seemingly unacceptable overhead does not have a significant impact on the overall performance of programs. This is attributed to the interleaving of data-in-code reads with numerous other instructions, rendering the overall overhead negligible. A more in-depth analysis of this performance impact will be presented in §VII-C.

Table 5: Context switch time (in μs). Smaller is better.

Kernel	2p	2p	2p	8p	8p	16p	16P
	0K	16K	64K	16K	64K	16K	64K
Standard	2.10	2.59	2.72	2.58	2.59	2.71	2.73
PXoM	2.13	2.61	2.70	2.64	2.65	2.72	2.71
Overhead	1.43%	0.77%	-0.74%	-0.93%	2.33%	0.37%	-0.73%

We need to store some metadata, such as MPK's PKey and embedded data list, in the process's context structure, which may cause overhead during context switches in the kernel. Table 5 shows the context switch time for both standard kernel and modified kernel. In the first row, the upper half displays the number of processes involved in the context switches, while the lower half shows the memory usage of each process. For instance, "2p/16k" represents a context switch between two processes, each of which uses 16 KB of memory. All entries exhibit overhead values that are clustered around zero, indicating that PXoM does not have a significant impact on the performance of kernel context switches. We present other lmbench results with low correlation to PXoM in a longer version of this paper (see Appendix C).

B. Macrobenchmarks

We evaluate the performance impact of PXoM on compute-intensive programs using SPEC CPU 2017, the latest generation of SPEC CPU benchmarks with larger and more

complex workloads than its predecessors. We compiled both the standard version of SPEC CPU 2017 and the version that was protected by PXoM, and ran them using the *ref* workload on our test machine. We take the running time of standard versions as the baseline to measure the additional overhead incurred by PXoM's protection. Besides, to compare with the performance data of previous XoM approaches, we also conducted a performance evaluation on SPEC CPU 2006 (see Appendix C).

Figure 7 shows the runtime slowdown caused by PXoM on SPEC CPU 2017, with the green and blue striped bars on the rightmost side showing the average and geometric mean value of overhead, respectively. From Figure 7, we can see that eleven overhead values are very close to zero, while five benchmarks (ID numbers: 523, 623, 511, 500, and 600) reveal relatively high overhead. The peak overhead value happens in `523.xalancbmk`. The two `xalancbmk` benchmarks (523 & 623) transform XML documents into HTML, text, or other XML document types. The `511.povray` is a ray-tracing program, and the two `perlbench` benchmarks (500 & 600) are lightweight Perl interpreters. As all these five benchmarks contain a lot of switch-loop structures, we conjecture that frequently reading the jump table to call small handler functions contributes to the larger overhead than the remaining benchmarks. Nonetheless, the average and geometric mean overhead of tested SPEC benchmarks are 0.36% and 0.25%, respectively, indicating that PXoM introduces a negligible performance impact on CPU-intensive programs. In addition to SPEC CPU 2017, we also demonstrate that PXoM introduces minimal runtime overhead to mainstreams web servers and databases (see Appendix C).

However, we encountered a major challenge in the current inability to reproduce or replicate previous XoM results, which is an issue that unfortunately plagues the security field. None of the preceding XoM studies, to the best of our knowledge, have made their tools publicly available. We have conducted a separate experiment on SPEC CPU 2006 in order to compare the performance data of PXoM with that reported by other prominent peer tools in their respective papers [23]–[25], [27], [29], [31], [32]. Please see Appendix C for details. In summary, PXoM still exhibits the lowest overhead (0.30%) among all compared XoM prototypes.

C. Overhead Causation Analysis

The performance overhead of PXoM is predominantly influenced by two key factors: 1) size of the embedded data list; and 2) frequency of data-in-code reads. We introduce the term "read intensity" to denote the ratio of data-in-code read instructions to the total number of executed instructions. Next, we conduct a quantitative examination of these two factors and elucidate the rationale behind the observed negligible overhead incurred by PXoM.

Read Latency Figure 8 illustrates the time taken for performing different numbers of data-in-code reads under different embedded data list sizes (N). The horizontal axis denotes the number of data-in-code reads, while the vertical axis represents the time taken to complete the specified number of read requests. In the "Normal" case, which corresponds to disabling PXoM protection, the completion time for read requests remains relatively consistent despite an increase in the amount of reads. However, in cases where PXoM is activated, an increased volume of read requests correlates with a more pronounced overhead. Moreover, a larger size of the embedded data list contributes to a heightened level of overhead. This observation highlights that when a significant portion of a program's instructions is dedicated to data-in-code reads, the overhead becomes prominent, especially with larger values of N. However, in real-world programs, data-in-code read instructions are typically interspersed among a multitude of other instructions. Furthermore, as shown in Tables 2 and Table 3, the average size of N (embedded data list size) for both open-source and COTS programs is relatively small, with average values of $4,290$ and $1,217$, respectively. The consequence is that PXoM incurs minimal runtime overhead in real-world programs, as evidenced by the performance data of macrobenchmarks. Next, we will quantitatively evaluate the performance impact caused by read intensity.

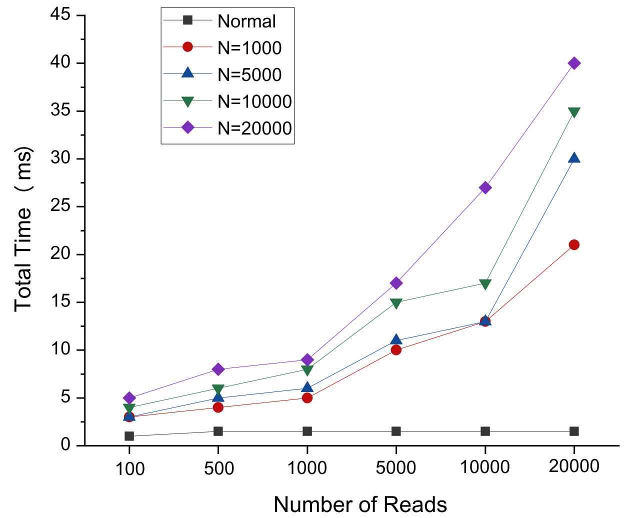

Figure 8: The time for different numbers of embedded data reads for a normal program and PXoM enabled programs with different embedded data list size. The N in the figure means the size of the embedded data list.

Read Intensity As shown in Figure 8, there is a direct relationship between a program's intensity of data-in-code reads and the resulting overhead. To capture this, we define the term "Read Intensity," as given by Equation 3.

$$Read\ Intensity = \frac{\#\ of\ Read\ Requests}{\#\ of\ Executed\ Instructions} \quad (3)$$

We have gathered statistics on data-in-code reads and the total number of executed instructions during performance evaluations for SPEC CPU 2017, webservers, databases, and OpenSSL. We have calculated their respective Read Intensity values, as depicted in Figure 9. The program with the lowest Read Intensity is databases, at 4.8E^{-12}, indicating that it performs one data-in-code read for every hundred billion

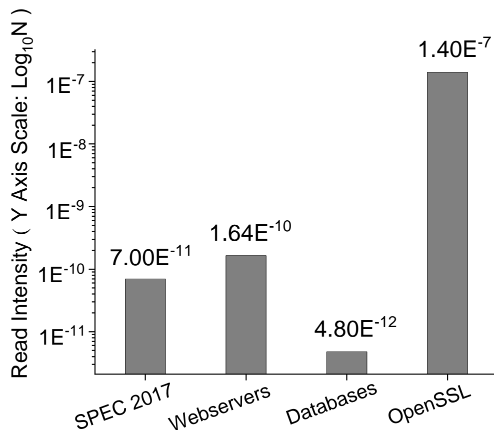

Figure 9: Read intensity of difference benchmarks. The Y-axis values represent the proportion of data-in-code instructions relative to all executed instructions.

instructions on average. OpenSSL embeds a significant amount of data within the code region to enhance the performance of cryptography algorithms. It exhibits the highest Read Intensity at $1.4E^{-7}$. Nevertheless, even in OpenSSL, on average, it takes a million instructions to perform one data-in-code read. The average Read Intensity for all programs is $3.51E^{-8}$, signifying one data-in-code read is performed after executing ten million instructions on average. This illustrates that while data-in-code reads are not rare in practice, their occurrence rate is extremely low, resulting in PXoM's practical overhead being negligible.

VIII. Discussion & Conclusion

Kernel Component Security Adding code to the trusted computing base is risky, so we need to pay extra attention to ensuring the security of any additions made to the kernel. The first potential threat is the user-controllable ".xom" section. When the kernel loads a binary, it parses the contents of the ".xom" section into the kernel's structures. Improper checks during this process could allow an attacker to exploit the kernel. Therefore, we must conduct careful and strict checks when parsing this list to prevent buffer overflow attacks. In addition, we added some pointers in the kernel to store data related to XoM. When using these pointers, strict checks must also be enforced to prevent vulnerabilities such as use-after-free and double-free. Another potential threat is the race condition between different threads. When embedded data reading is allowed, the target code page will be in a readable state for a very short time. If control is taken over by another thread at this time, that thread may disclose the readable memory page. Therefore, it should be ensured that the permission for the reading operation is atomic, and control cannot be taken away during this operation.

Protection on Dynamically Loaded Code For now, PXoM is not designed to protect dynamically loaded (dlopen) code and dynamically generated code, such as in a program running JavaScript code in a JIT engine. Achieving protection for them will be our future work. For the dynamically loaded code, the only difference is the loading process. We will hook the GNU C Library to protect dynamically loaded code. For the dynamically generated code, when the JIT engine generates JIT compiled code, we can know the location of all embedded data, so we can mark these locations as readable in the JIT engine and apply PXoM protection to the generated code.

MPK Security While Memory Protection Keys (MPK) provides an efficient mechanism for managing memory permissions, it also raises concerns regarding its own security pitfalls [84]. Fortunately, new defensive strategies have emerged to further strengthen the security of MPK [85], [86]. Ongoing improvements and refinements in this evolving domain continue to enhance the security of MPK. Many existing works have utilized MPK for sensitive data isolation. For instance, ERIM [87] and Hodor [88] utilize MPK to isolate sensitive data and only allow trusted code to access it by controlling read permissions to these sensitive data areas. Similarly, Burow et al. [89] investigate using MPK to provide stronger guarantees for shadow stacks, which are used to make sensitive data on the stack tamper-resistant [90]. Additionally, Jin et al. [91] employ the MPK mechanism to safeguard sensitive key-related data in cryptographic algorithm implementations.

Conclusion Execute-only memory (XoM) is a promising solution to prevent memory disclosure and counter JIT-ROP attacks. This paper presents PXoM, a technique that retrofits XoM into stripped binaries without embedded data relocation. Unlike existing approaches, PXoMenables fine-grained memory permission control within a memory page without requiring compile-time transformations or binary patching. Performance evaluations on large programs show negligible runtime overhead, and security assessments suggest PXoMis viable for real-world adoption, potentially shifting the memory defense landscape in favor of defenders.

Acknowledgment

We thank our shepherd and all the anonymous reviewers for their valuable comments to improve this paper. This work is supported by the National Nature Science Foundation of China under Grant No. 62272351, 61972297, and 62172308. Jiang Ming was supported by NSF grants 2312185 & 2417055 and Google Research Scholar Award.

References

[1] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P'13)*, 2013.

[2] Volodymyr Kuznetzov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, 2014.

[3] Christian DeLozier, Kavya Lakshminarayanan, Gilles Pokam, and Joseph Devietti. Hurdle: Securing Jump Instructions Against Code Reuse Attacks. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*, 2020.

[4] Victor Duta, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. PIBE: Practical Kernel Control-Flow Hardening with Profile-Guided Indirect Branch Elimination. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'21)*, 2021.

[5] Haehyun Cho, Jinbum Park, Adam Oest, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. ViK: Practical Mitigation of Temporal Memory Safety Violations through Object ID Inspection. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*, 2022.

[6] Haotian Zhang, Mengfei Ren, Yu Lei, and Jiang Ming. One Size Does Not Fit All: Security Hardening of MIPS Embedded Systems via Static Binary Debloating for Shared Libraries. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*, 2022.

[7] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*, 2007.

[8] Michael Backes and Stefan Nürnberger. Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In *Proceedings of the 23rd USENIX Conference on Security Symposium (USENIX Security'14)*, 2014.

[9] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15)*, 2015.

[10] PaX Team. Address Space Layout Randomization (ASLR). https://pax.grsecurity.net/docs/aslr.txt, 2003.

[11] Sandeep Bhatkar, Daniel C DuVarney, and Ron Sekar. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the 12th Conference on USENIX Security Symposium (USENIX Security'03)*, 2003.

[12] Sandeep Bhatkar, Daniel C DuVarney, and R Sekar. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *Proceedings of the 14th Conference on USENIX Security Symposium (USENIX Security'05)*, 2005.

[13] Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *Proceedings of the 21st Conference on USENIX Security Symposium (USENIX Security'12)*, 2012.

[14] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W Davidson. ILR: Where'd My Gadgets Go? In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P'12)*, 2012.

[15] Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. Profile-guided Automated Software Diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'13)*, 2013.

[16] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06)*, 2006.

[17] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P'12)*, 2012.

[18] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS'12)*, 2012.

[19] David Williams-King, Graham Gobieski, Kent Williams-King, James P Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P Kemerlis, Junfeng Yang, and William Aiello. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, 2016.

[20] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*, pages 1–8, 2009.

[21] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P'13)*, 2013.

[22] Salman Ahmed, Ya Xiao, Kevin Z Snow, Gang Tan, Fabian Monrose, and Danfeng Yao. Methodologies for Quantifying (Re-)randomization Security and Timing under JIT-ROP. In *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS'20)*, 2020.

[23] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS'14)*, 2014.

[24] Kjell Braden, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Christopher Liebchen, and Ahmad-Reza Sadeghi. Leakage-Resilient Layout Randomization for Mobile Devices. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS'16)*, 2016.

[25] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*, pages 763–780. IEEE, 2015.

[26] Donghyun Kwon, Jangseop Shin, Giyeol Kim, Byoungyoung Lee, Yeongpil Cho, and Yunheung Paek. uXOM: Efficient eXecute-Only Memory on ARM Cortex-M. In *Proceedings of the 28th Conference on USENIX Security Symposium (USENIX Security'19)*, 2019.

[27] Jason Gionta, William Enck, and Peng Ning. HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY'15)*, 2015.

[28] Yaohui Chen, Dongli Zhang, Ruowen Wang, Rui Qiao, Ahmed M Azab, Long Lu, Hayawardh Vijayakumar, and Wenbo Shen. NORAX: Enabling Execute-Only Memory for COTS Binaries on AArch64. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P'17)*, 2017.

[29] Mingwei Zhang, Michalis Polychronakis, and R. Sekar. Protecting COTS Binaries from Disclosure-Guided Code Reuse Attacks. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC '17)*, 2017.

[30] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler Optimizations for Improving Data Locality. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, 1994.

[31] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. Heisenbyte: Thwarting Memory Disclosure Attacks using Destructive Code Reads. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS'15)*, 2015.

[32] Jan Werner, George Baltas, Rob Dallara, Nathan Otterness, Kevin Z Snow, Fabian Monrose, and Michalis Polychronakis. No-Execute-After-Read: Preventing Code Disclosure in Commodity Software. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIACCS'16)*, 2016.

[33] Kevin Z Snow, Roman Rogowski, Jan Werner, Hyungjoon Koo, Fabian Monrose, and Michalis Polychronakis. Return to the Zombie Gadgets: Undermining Destructive Code Reads via Code Inference Attacks. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P'16)*, 2016.

[34] Hyungseok Kim, Soomin Kim, Junoh Lee, Kangkook Jee, and Sang Kil Cha. Reassembly is Hard: A Reflection on Challenges and Strategies. In *Proceedings of the 32nd USENIX Conference on Security Symposium (USENIX Security'23)*, 2023.

[35] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual. https://intel.ly/3U3Av2E, [online].

[36] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC'19)*, 2019.

[37] Chengbin Pang and Ruotong Yu and Yaohui Chen and Eric Koskinen and Georgios Portokalidis and Bing Mao and Jun Xu. SoK: All You Ever Wanted to Know About x86/x64 Binary Disassembly But Were Afraid to Ask. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P'21)*, 2021.

[38] Chengbin Pang, Tiantai Zhang, Ruotong Yu, Bing Mao, and Jun Xu. Ground Truth for Binary Disassembly is Not Easy. In *Proceedings of the 31st USENIX Security Symposium (USENIX'22)*, 2022.

[39] Larry McVoy. LMbench - Tools for Performance Analysis. https://lmbench.sourceforge.net/, [online].

[40] Standard Performance Evaluation Corporation. SPEC CPU 2006. https://www.spec.org/cpu2006/, [online].

[41] Standard Performance Evaluation Corporation. SPEC CPU® 2017. https://www.spec.org/cpu2017/, [online].

[42] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Transactions on Information and System Security*, 15(1), March 2012.

[43] R. M. Graham. Multics System-Programmers' Manual. https://bit.ly/488lNgQ, July 1967.

[44] Siguza. PAN. https://blog.siguza.net/PAN/, [online].

[45] Android. Execute-Only Memory (XOM) for AArch64 Binaries. https://source.android.com/docs/security/test/execute-only-memory, [online].

[46] Dave Hansen. X86, Pkeys: Execute-Only Support. https://lore.kernel.org/linux-mm/20160212210240.CB4BB5CA@viggo.jf.intel.com/, [online].

[47] Rick Edgecombe. Touch But Don't Look - Running the Kernel in Execute-Only Memory. https://bit.ly/4h2YzfW, [online].

[48] Theo de Raadt. Synthetic Memory Protections - An Update on ROP Mitigations. https://www.openbsd.org/papers/csw2023.pdf, [online].

[49] Xiaozhu Meng and Barton P. Miller. Binary Code is Not Easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16)*, 2016.

[50] Manuel Rigger, Stefan Marr, Stephen Kell, David Leopoldseder, and Hanspeter Mössenböck. An Analysis of x86-64 Inline Assembly in C Programs. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '18)*, 2018.

[51] Jonathan Corbet. A new LLVM CFI implementation. https://lwn.net/Articles/898040/, [online].

[52] Richard Wartell, Yan Zhou, Kevin W Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. Differentiating Code from Data in x86 Binaries. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, 2011.

[53] Zhe Wang, Chenggang Wu, Mengyao Xie, Yinqian Zhang, Kangjie Lu, Xiaofeng Zhang, Yuanming Lai, Yan Kang, and Min Yang. SEIMI: Efficient and Secure SMAP-Enabled Intra-process Memory Isolation. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[54] Gerald J. Popek and David A. Farber. A model for verification of data security in operating systems. In *Communications of the ACM*, 1978.

[55] XIAO GUANGRONG. Linux Kernel 2.6.31 - 'perf_counter_open()' Local Buffer Overflow. https://www.exploit-db.com/exploits/33228, [online].

[56] Anonymous. Nvidia Linux Driver - Local Privilege Escalation. https://www.exploit-db.com/exploits/20201, [online].

[57] Enrico Perla and Massimiliano Oldani. *A Guide to Kernel Exploitation: Attacking the Core*. Elsevier, 2010.

[58] Jason Gionta, William Enck, and Per Larsen. Preventing Kernel Code-Reuse Attacks Through Disclosure Resistant Code Diversification. In *Proceedings of 2016 IEEE Conference on Communications and Network Security (CNS'16)*, 2016.

[59] Marios Pomonis, Theofilos Petsios, Angelos D Keromytis, Michalis Polychronakis, and Vasileios P Kemerlis. kRˆX: Comprehensive Kernel Protection Against Just-In-Time Code Reuse. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys'17)*, 2017.

[60] Michael Larabel. Intel MPX Support Will Be Removed From Linux—Memory Protection Extensions Appear Dead. https://www.phoronix.com/news/Intel-MPX-Kernel-Removal-Patch, 2018.

[61] Spyridoula Gravani, Mohammad Hedayati, John Criswell, and Michael L Scott. Fast Intra-kernel Isolation and Security with IskiOS. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'21)*, 2021.

[62] Qualcomm Technologies. Pointer Authentication on ARMv8.3. https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf, [online].

[63] Intel. Control Flow Enforcement Technology (CET). https://www.intel.com/content/dam/develop/external/us/en/documents/catc17-introduction-intel-cet-844137.pdf, [online].

[64] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (S&P '14)*, pages 575–589. IEEE, 2014.

[65] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security '15)*, pages 161–176, 2015.

[66] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security '14)*, pages 401–416, 2014.

[67] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*, pages 901–913, 2015.

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

[69] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security '14)*, pages 417–432, 2014.

[70] Sabine Houy and Alexandre Bartel. Lessons Learned and Challenges of Deploying Control Flow Integrity in Complex Software: The Case of OpenJDK's Java Virtual Machine. In *Proceedings of the 2024 IEEE Secure Development Conference (SecDev '24)*, 2024.

[71] Arif Ali Mughal. The Art of Cybersecurity: Defense in Depth Strategy for Robust Protection. *International Journal of Intelligent Automation and Computing*, 1(1):1–20, 2018.

[72] Sheng Yu, Yu Qu, Xunchao Hu, and Heng Yin. DeepDi: Learning a Relational Graph Convolutional Network Model on Instructions for Fast and Accurate Disassembly. In *Proceedings of the 31st USENIX Security Symposium (USENIX '22)*, 2022.

[73] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. Probabilistic Disassembly. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*, 2019.

[74] Antonio Flores-Montoya and Eric Schulte. Datalog Disassembly. In *Proceedings of the 29th USENIX Conference on Security Symposium (USENIX Security'20)*, 2020.

[75] Dennis Andriesse, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *Proceedings of the 25th USENIX security symposium (USENIX security '16)*, pages 583–600, 2016.

[76] Erick Bauman, Zhiqiang Lin, Kevin W Hamlen, et al. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. In *Proceedings of the 2018 Network and Distributed Systems Security Symposium (NDSS '18)*, 2018.

[77] Kexin Pei, Jonas Guan, David Williams-King, Junfeng Yang, and Suman Jana. XDA: Accurate, Robust Disassembly with Transfer Learning. In *Proceedings of the 2021 Network and Distributed Systems Security Symposium (NDSS '21)*, 2021.

[78] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. Disassembly of Executable Code Revisited. In *Proceedings of 9th Working Conference on Reverse Engineering*, 2002.

[79] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. Egalito: Layout-Agnostic Binary Recompilation. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*, 2020.

[80] Jonathan Salwan. ROPgadget. https://github.com/JonathanSalwan/ROPgadget, [online].

[81] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Conference on Security Symposium (USENIX Security'11)*, 2011.

[82] Andrei Homescu, Michael Stewart, Per Larsen, Stefan Brunthaler, and Michael Franz. Microgadgets: Size Does Matter in Turing-Complete Return-Oriented Programming. In *Proceedings of the 6th USENIX Workshop on Offensive Technologies (WOOT '12)*, 2012.

[83] Salman Ahmed. JITROP-Native. https://github.com/salmanyam/jitrop-native, [online].

[84] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security '20)*, 2020.

[85] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security '22)*, 2022.

[86] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. You Shall Not (By)pass! Practical, Secure, and Fast PKU-based Sandboxing. In *Proceedings of the 17th European Conference on Computer Systems (EuroSys '22)*, 2022.
[87] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *Proceedings of the 28th Conference on USENIX Security Symposium (USENIX Security'19)*, 2019.
[88] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC'19)*, 2019.
[89] Nathan Burow, Xinping Zhang, and Mathias Payer. SoK: Shining Light on Shadow Stacks. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.
[90] Thurston HY Dang, Petros Maniatis, and David Wagner. The Performance Cost of Shadow Stacks and Stack Canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIA CCS'15)*, 2015.
[91] Xuancheng Jin, Xuangan Xiao, Songlin Jia, Wang Gao, Hang Zhang, Dawu Gu, Siqi Ma, Zhiyun Qian, and Juanru Li. Annotating, Tracking, and Protecting Cryptographic Secrets with CryptoMPK. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (S&P'22)*, 2022.
[92] The MITRE Corporation. CVE-2013-2028 Detail. https://www.cve.org/CVERecord?id=CVE-2013-2028, [online].
[93] Brian Johannesmeyer, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical Data-Only Attack Generation. In *Proceedings of the 33rd USENIX Conference on Security Symposium (USENIX Security' 24)*, 2024.

APPENDIX

A. Case Studies for Security Evaluation

Exploit Memory Disclosure Vulnerability We use the CVE-2013-2028 [92], a Nginx arbitrary memory disclosure vulnerability, to showcase PXoM's effectiveness. This vulnerability is a potent stack overflow that enables an attacker to carry out arbitrary memory reads. We apply PXoM to a vulnerable binary version of Nginx and run it as a web server. After that, we modify the JIT-ROP attack framework, jitrop-native [83], to specifically adapt it to the CVE-2013-2028, and use it to trigger this vulnerability and dynamically search for gadgets. The attack is detected and prevented when the exploit tries to reveal the first code byte, indicating that PXoM is capable of safeguarding Nginx from memory disclosure.

Construct WRPKRU Gadgets The value of PKRU can be changed using the *WRPKRU* instruction at the user level. Intuitively, if this instruction is located by attackers in executable data blocks, they can use it to regain read permission for the code pages. In order to successfully change the permission of a page group using the WRPKRU instruction, four operations must be performed: 1) storing the permission value to EAX; 2) writing zero to ECX; 3) writing zero to EDX; and 4) executing WRPKRU. Please note that when executing the WRPKRU instruction, EAX stores the permission value for all page groups that will later be written into PKRU, and the values of ECX and EDX must be zero to avoid a general-protection exception (#GP). Completing these operations may require more than four gadgets. For instance, in OpenSSL, only `XCHG` instructions can change `EAX`'s value, such as `XCHG EDI, EAX`. Thus, an additional gadget is necessary to write the permission value to `EDI`, which can then be swapped with EAX using `XCHG`. To find gadgets capable of completing these operations, we conduct a gadget search on all binaries' executable data in the Pang et al.'s data set [38]. The dataset revealed that only 23 binaries' executable data contain gadgets that can complete one or two operations, but no binary gadgets that can complete all four operations. Interestingly, even treating each byte as an opcode, we only found 26 WRPKRU instructions in the $\sim 20GB$ dataset, and none of them was in the executable data areas. This rarity of the byte sequence of WRPKRU (`0F 01 EF`) in the compiled binary could explain the difficulty of finding gadgets capable of performing all four operations.

Attacker-Controllable Syscalls Despite the fact that attackers may attempt to leverage system calls (e.g., *mprotect* and *execve*) to conduct their second-stage attack, thereby circumventing the necessity for Turing-complete gadgets and minimizing gadget requirements, they still necessitate multiple gadgets to manipulate the parameters of these system calls. Johannesmeyer et al. [93] enumerated twelve system calls that could potentially be exploited to implement such attacks. We conducted an additional experiment to search for gadgets capable of manipulating the parameters of these twelve system calls within the embedded data regions of the dataset presented in Table 2, and no such gadgets were discovered.

B. Case Studies of Embedded Data in COTS Binaries

Through empirical study, we categorized the embedded data into the following four types:

1) Embedded Constants: Independent constants dispersed throughout the program, each referenced separately. These constants can include integers, floating-point numbers, or other data structures.
2) Embedded Arrays: Groups of constants organized into arrays, where each element is accessed using an "`array pointer + index`."
3) Embedded Strings: Strings embedded directly within the code section.
4) Jump Tables: Tables that store target addresses for switch-case structures.

1) Embedded Constants: Using Skype version 8.129.0.202 as an example, Figure A1 illustrates the embedded constants within the binary file *skypeforlinux*. In lines 1 and 3, two 128-bit integers are embedded, with the `paddd` instruction being used in lines 10 and 12 to add them to the value in the `xmm0` register. Similarly, in lines 5 and 7, two 256-bit integers are embedded, which are then used in lines 14 and 17.

2) Embedded Arrays: Unlike embedded constants, where each constant has its own reference, embedded arrays group constants together, with each element accessed via an array pointer. For example, in the main binary *resolve* of DaVinci Resolve (version 19.0.1), there is an embedded array, as shown in Figure A2. In line 9, an array of 4,160 bytes is embedded, and in line 1, its reference is loaded into the `rbp` register. Lines 3, 5, and 6 show how values from the array are accessed using the array pointer stored in the `rbp` register, with the `rsi` register serving as the index. The retrieved values are then loaded into the `r8` and `r9` registers.

```
1  .text:7D642  lea  rsi, jpt_7D666
2  .text:7D649  mov  r10, rdi
3  .text:7D64C  neg  r10
4  .text:7D64F  add  r10, 40h
5  .text:7D653  and  r10, 3Fh  ;Switch 64 cases
6  ...
7  .text:7D666  jmp  rsi       ;Switch Jump
8  ...
9  jpt_7D666:
10 .text:7D8C0  dq 7D8C0h - offset loc_7D726,
11                 7D8C0h - offset loc_7D723
12 .text:7D8D0  dq 7D8C0h - offset loc_7D72B,
13                 7D8C0h - offset loc_7D734
14 .text:7D8E0  dq 7D8C0h - offset loc_7D741,
15                 7D8C0h - offset loc_7D749
16 .text:7D8F0  dq 7D8C0h - offset loc_7D749,
17                 7D8C0h - offset loc_7D749
18 ...
```

Figure A4: A jump table in the *dpcpp* of Intel DPC++.

```
1  .text:1A913A0 xmmword_1A913A0 xmmword
2                3000000020000000100000000h
3  .text:1A913B0 xmmword_1A913B0 xmmword
4                4000000040000000400000004h
5  .text:1A913C0 ymmword_1A913C0 ymmword
6                0000000002000000040...(256 Bits)
7  .text:1A913E0 ymmword_1A913E0 ymmword
8                0800000008000000080...(256 Bits)
9  ...
10 .text:1A91C3F paddd xmm0, cs:xmmword_1A913A0
11 ...
12 .text:1A91CDB paddd xmm0, cs:xmmword_1A913B0
13 ...
14 .text:1A92631 vpaddd ymm4, ymm4,
15                      cs:ymmword_1A913C0
16 ...
17 .text:1A926AE vpaddd ymm4, ymm4,
18                      cs:ymmword_1A913E0
19 ...
```

Figure A1: Embedded constants in the *skypeforlinux* of Skype.

```
1  .text:A5DEFF2 lea rbp, qword_A5DF8C0;Array Ref
2  ...
3  .text:A5DF0A0 mov r8, [rbp+rsi*8+1000h]
4  ...           ; Get values from an array
5  .text:A5DF0C6 xor r8, [rbp+rsi*8+0]
6  .text:A5DF0CB mov r9, [rbp+rdi*8+7]
7  ...
8  qword_A5DF8C0:   ; An array of 4160 bytes
9  .text:A5DF8C0 dq 2 dup(0D83078C018601818h),
10                  2 dup(2646AF05238C2323h)
11 .text:A5DF8E0 dq 2 dup(0B891F97EC63FC6C6h),
12                  2 dup(0FBCD6F13E887E8E8h)
13 .text:A5DF900 dq 2 dup(0CB13A14C87268787h),
14                  2 dup(116D62A9B8DAB8B8h)
15 ...
```

Figure A2: An embedded array in the *resolve* of DaVinci.

3) Embedded Strings: Embedded strings are a specific type of embedded constant, characterized by their variable size and termination with a 0x00 byte. Figure A3 provides an example of embedded strings found in the *teamviewerd* daemon of TeamViewer (version 15.58.4). In lines 6, 7, and 8, three strings are embedded, with a reference to the string embedded in line 6 made in line 1.

```
1 .text:A748F4  lea rax, aRc48xInt ;String Ref
2 .text:A748FB  mov edx, dword cs:qword_16781C0
3 .text:A74901  bt  edx, 14h
4 .text:A74905  jb  short loc_1F3853
5 ...
6 .text:A74940 aRc48xInt  db 'rc4(8x,int)',0
7 .text:A7494C aRc48xChar db 'rc4(8x,char)',0
8 .text:A74959 aRc416xInt db 'rc4(16x,int)',0
9 ...
```

Figure A3: Embedded strings in the *teamviewerd* daemon of TeamViewer.

4) Jump Tables: A jump table is a specialized type of embedded array and one of the most common data structures embedded in code. It stores the target addresses for switch-case structures. In Position Independent Code (PIC), the jump table holds the offset between the target code and the jump instruction, while in non-PIC code, it stores the absolute address of the target code. Although compilers like GCC and LLVM typically place jump tables in the data segment, some compilers, such as Intel's C++ compiler, prefer to embed jump tables closer to the code that uses them. This approach reduces the likelihood of cache misses, thereby improving program performance. In contrast, placing the jump table in a distant data segment can increase the frequency of cache misses.

Figure A4 provides an example of a jump table in the *dpcpp* binary of Intel DPC++ (version 2.1.79). At line 10, a switch structure with 64 cases is defined. Since this binary is a position-independent executable (PIE), the jump table starting at line 10 stores offsets of the target addresses relative to line 7. In line 1, the address of the jump table is loaded into the `rsi` register, and after performing a series of calculations based on the index value, the target address is determined. Finally, in line 7, the program jumps to the calculated target address.

C. Longer Version

For algorithm of Unidirectional Disassembly, performance on real-world applications, other microbenchmark results, performance comparison, and high-frequency data read optimization evaluation, please refer to the longer version of this paper at https://arxiv.org/abs/2412.02110.

Artifact Appendix

A. Description & Requirements

In our paper, we present PXoM, a hardware-assisted approach to retrofitting XoM (Execute-only Memory) for stripped binaries, without the need for relocating embedded data. PXoM is a comprehensive system that includes a full-stack toolchain, from user-level applications to a custom kernel, designed to provide XoM protection for programs while ensuring compatibility with legitimate embedded data reads within code sections, all without the need for relocating embedded data.

In this artifact, we provide the following:

1) Virtual Machine (PXoM_Artifact.ova): An out-of-the-box (OOB) virtual machine with a customized system kernel and user-space toolchain pre-deployed for easy access to PXoM. This VM offers a convenient way to quickly start testing PXoM on various programs and reproducing the evaluations presented in this paper.
2) Source Code (PXoM_Artifact-0.1.tar.gz): The source code for the PXoM kernel, user-space toolchain, and all the experiments described in this artifact.
3) Documentation (PXoM_Artifact.pdf): Detailed instructions on how to use the PXoM virtual machine and the workflow for conducting the experiments.

In this artifact appendix, we will outline the hardware and software requirements for PXoM, the steps to install the virtual machine, the major claims from our paper, and the experimental workflows.

How to access

PXoM Virtual Machine: 10.5281/zenodo.13892220

Source Code: 10.5281/zenodo.14251050

PXoM Artifact Documentation: 10.5281/zenodo.14251155

Hardware dependencies

The only required hardware feature is MPK, which is supported by the following CPUs:

1) Intel® Desktop CPUs, Comet Lake (10th Gen Core™) and later;
2) Intel® Server CPUs, Xeon® Skylake and later;
3) AMD Desktop CPUs, Ryzen™ 5000 and later;
4) AMD Server CPUs, EPYC™ Milan (7003 Series) and later.

Software dependencies

There are two options for deploying the PXoM VM:

1) On a host running Ubuntu 22.04 or later, you can import the PXoM virtual machine using VMWare Workstation Pro 17.6.0 or higher. Please note that MPK is not supported on Windows hosts, so VMWare Workstation Pro must be installed on a Linux-based host.
2) Directly import the PXoM virtual machine on a machine running ESXi 8.0 Update 3 or later.

Benchmarks

Most of the benchmarks are included with the PXoM VM image. However, we have excluded Pang et al.'s dataset from the image due to its large size (∼56GB after decompression), which would make the image excessively bloated. You can obtain Pang et al.'s dataset from their Github repository. Please refer to the instructions in the artifact documentation (PXoM_Artifact.pdf) before obtaining the dataset.

B. Artifact Installation & Configuration

The only installation step is to import the PXoM virtual machine (PXoM_Artifact.ova) into VMWare Workstation Pro or VMWare ESXi. All the experiments from our paper can be conducted within this virtual machine.

For instructions on how to import the OVA file into VMWare Workstation Pro and VMWare ESXi, please refer to the VMware Workstation documentation and VMware ESXi documentation. After importing the PXoM virtual machine, you can adjust its memory size and the number of CPUs. We recommend allocating more than 32GB of memory and assigning more than 10 cores.

Optional: To compile the PXoM kernel on a bare-metal machine, please follow the same process as you would for the standard Linux kernel. For example, begin with the instructions starting at Step 3 in this guide: https://phoenixnap.com/kb/build-linux-kernel.

C. Major Claims

Our paper makes two major claims:

- (C1): PXoM can protect stripped binaries from JIT-ROP attacks while allowing legitimate embedded data reads, without requiring relocating embedded data. This is demonstrated through experiments (E1) and (E2).
- (C2): PXoM introduces negligible performance overhead. This is validated by experiments conducted on lmbench (E3), SPEC CPU 2017 (E4), Webservers (E5), and Databases (E6).

D. Evaluation

The experiments are divided into six parts: (E1): JIT-ROP Defense Demonstration; (E2): Disassembly Result Evaluation; (E3): lmbench; (E4): SPEC CPU 2017; (E5): Web Servers; and (E6): Databases. E1 and E2 support C1, while E3∼E6 support C2. We provide instructions to reproduce the experiments described in our paper; however, we do not claim the "reproduced" badge for two reasons:

1) The dataset for E2 is too large, requiring 5 to 6 days to fully evaluate the entire dataset.
2) The virtual machine provides the most reliable environment to ensure PXoM functions correctly by masking hardware differences, which ensures proper operation across various devices. However, virtualization may lead to inaccurate performance evaluation results. Moreover, Intel's hybrid architecture of E-Cores and P-Cores can further amplify experimental inaccuracies.

The `~/PXoM_Artifact/experiments` folder within the virtual machine contains all the experiments. Please conduct each experiment in its corresponding folder.

*1) **Experiment (E1):*** [JIT-ROP Defense Demonstration] [20 human-minutes + 5 compute-minutes]: Demonstrating how PXoM protects programs from JIT-ROP attacks while allowing legitimate embedded data reads.

[*Workflow*]

1. Run the vulnerable demo server:

```
> sudo ./cipher_helper
```

2. Exploit the vulnerable server:

```
> python exploit.py
```

3. Protect the vulnerable server using the PXoM toolchain:

```
> pxom protect –i cipher_server –o cipher_server.xom
```

4. Run the protected server and attempt the exploit again:

```
> sudo ./cipher_helper.xom
> python exploit.py
```

5. The exploitation will fail, and you can check the kernel log for details:

```
> sudo dmesg
```

*2) **Experiment (E2):*** [Disassembly Result Evaluation] [15 human-minutes + 10 compute-minutes (5∼6 days for the entire dataset)]: Evaluating the effectiveness of our disassembly strategy.

[*Workflow*]

1. Protect the program and print the embedded data list:

```
> pxom protect –i openssl_O3 –o openssl_O3.xom
> pxom print –i openssl_O3.xom > xom_edata
```

2. Extract the ground truth using Pang et al.'s toolchain:

```
> objcopy ––dump–section .rand=tmp_gt.gz openssl_O3
&& gzip –d tmp_gt.gz
> python extract_gt/extract_edata.py –b openssl_O3 –m
tmp_gt –o tmp_pb –L gt_edata
```

3. Compare the disassembly results with the ground truth:

```
> python compare_results.py openssl_O3 xom_edata
gt_edata
```

[*Results*] Code Coverage and Overall Coverage for binaries.

*3) **Experiment (E3):*** [lmbench] [10 human-minutes + 20 compute-minutes]: Evaluating the performance overhead of PXoM's kernel modifications.

[*Workflow*]

1. Run lmbench:

```
> make results
```

2. Compare the results with the baseline results.

[*Results*] Performance overhead of kernel modification.

*4) **Experiment (E4):*** [SPEC CPU 2017] [20 human-minutes + 6 compute-hours]: Evaluating the performance overhead of PXoM's protection on compute-intensive programs.

[*Workflow*]

1. Run the standard SPEC CPU 2017:

```
> ./run_standard.sh
```

2. Run the PXoM-protected version of SPEC CPU 2017:

```
> ./run_pxom.sh
```

3. Compare the results:

```
> python compare.py
```

[*Results*] Performance overhead of PXoM on protecting SPEC CPU 2017

*5) **Experiment (E5):*** [Web Servers] [15 human-minutes + 10 compute-minutes]: Evaluating the performance overhead of PXoM on protecting web servers.

[*Workflow*]

For each web server, the workflow is the same:

1. Start the standard version of the web server:

```
> sudo ./start_standard
```

2. Obtain the baseline runtime:

```
> python run_standard.py
```

3. Stop the standard version of the web server:

```
> sudo ./stop_standard
```

4. Start the PXoM-protected web server:

```
> sudo ./start_pxom
```

5. Run the PXoM-protected tests:

```
> python run_pxom.py
```

6. Stop the PXoM-protected web server:

```
> sudo ./stop_pxom
```

7. Compare the results:

```
> python compare_results.py
```

[*Results*] Performance overhead of PXoM on protecting each web server.

*6) **Experiment (E6):*** [Databases] [20 human-minutes + 60 compute-minutes]: Evaluating the performance overhead of PXoM on protecting databases.

[*Workflow*]

For MySQL, MongoDB, and Redis:

1. Start the standard version of the database:

```
> ./start_standard
```

2. Obtain the baseline performance data:

```
> ./run_standard
```

3. Start the PXoM-protected database:

```
> ./start_pxom
```

4. Run the PXoM-protected tests:

```
> ./run_pxom
```

5. Compare the results:

```
> python compare_results.py
```

For SQLite:

1. Obtain the baseline performance data:

```
> ./run_standard
```

2. Run the PXoM-protected tests:

```
> ./run_pxom
```

3. Compare the results:

```
> python compare_results.py
```

[*Results*] Performance overhead of PXoM on protecting each database.