PLATYPUS: Restricting Cross-Module Transitions to Mitigate Code-Reuse Attacks

Apostolos Chatzianagnostou
CISPA Helmholtz Center for Information Security

Marcos Bajo
CISPA Helmholtz Center for Information Security

Christian Rossow
CISPA Helmholtz Center for Information Security

***Abstract*—Numerous techniques have been proposed to thwart code reuse attacks, yet practical adoption remains limited due to compatibility and deployment challenges. In the current and foreseeable Intel architecture landscape, the main line of defense against such attacks is Intel CET—a hardware-enforced control-flow integrity mechanism integrated into recent Intel x86-64 CPUs. However, despite its hardware-backed protections and widespread adoption, CET still provides only partial security: it continues to allow hijacked function pointers to invoke arbitrary functions *across* module boundaries, a capability that remains fundamental to many modern exploits.**

This paper proposes PLATYPUS, a novel defense on top of Intel CET to address this limitation. PLATYPUS enforces *execution jails* using lightweight address masking to ensure indirect control transfers remain within module boundaries. Cross-DSO function calls are only permitted via necessary PLT stubs specific to each DSO. The evaluation on our LLVM-based prototype, spanning 19 applications and 16 shared libraries (including glibc), demonstrates that PLATYPUS reduces indirectly accessible cross-DSO functions by over 98%. Performance testing with complex applications like Nginx and Redis shows that PLATYPUS incurs no more than 0.5% overhead.

1. Introduction

Ensuring the security of code written in memory-unsafe languages such as C and C++ is critical, since these languages lack built-in protections against memory errors. As a result, memory corruption vulnerabilities are widespread, presenting attackers with numerous avenues for exploitation. Among the various attack vectors, reusing code that is already present in a vulnerable program has emerged as one of the most prominent and enduring techniques. The potency of such code-reuse attacks (CRAs) and their ability to fully compromise a process have precipitated a long-standing arms race between defenders developing new mitigations and attackers devising novel bypass techniques.

Within this ongoing battle, Control Flow Integrity (CFI) [7] has been recognized as one of the most promising defensive mechanisms. The principal objective of CFI is to generate a Control Flow Graph (CFG) that represents all legitimate execution paths within a program. The CFG then guides the instrumentation of indirect control-flow transitions, which are the main hijacking targets, to permit only legitimate, precomputed transfers at runtime. Numerous CFI schemes have been proposed in recent years. These schemes are typically categorized based on the strictness of their CFGs as either coarse-grained or fine-grained, with the latter raising the bar for successful exploitation considerably. Despite extensive academic efforts to develop fine-grained CFI techniques, their adoption in practice remains notably limited [15]. A core reason for this is their lack of robust support for interoperability between protected and unprotected code [10], [46], a critical requirement in real-world environments where instrumented applications must frequently interact with third-party libraries.

Therefore, the vast majority of modern software relies on coarse-grained CFI schemes as the primary line of defense. The prevailing solutions, whose adoption is steadily increasing and is expected to become the default for future systems on the two most popular architectures are Intel's Control-Flow Enforcement Technology (CET) [40] and Arm's Branch Target Identification (BTI) [11], respectively. Both restrict the modularity of code-reuse gadgets to entire functions. This restriction is significant, as it complicates an attacker's ability to set up the necessary function arguments despite successfully hijacking control flow. Nonetheless, these protections remain insufficient: they do not prevent attackers from invoking arbitrary API functions in external libraries during a code pointer hijack. This limitation is severe as the misuse of many such functions can compromise vulnerable processes. For example, libc, the main interface for issuing system calls, features a rich set of functions that can be used as function-wise gadgets, exploitation dispatchers, or exploitation targets (e.g., `system()`). As a consequence, recent attacks [36], [44], [47], [64], [12] have leveraged these unrestrictedly reachable functions in cross-Dynamic Shared Object (DSO) transitions to bypass the widely deployed CET and BTI CFI schemes.

Seeing that arbitrary cross-DSO transitions are allowed is surprising. In Linux binaries, the vast majority of cross-DSO transitions occur through a well-defined interface: Procedure Linkage Tables (PLTs). PLTs have long been integral to dynamic linking in Linux systems, supporting address relocation and enabling features such as Address Space Layout Randomization (ASLR). Given this widely-established mechanism, it is contradictory that transfers between modules *bypassing* PLTs are permitted. In conjunction with Full RELRO mitigation, which is commonly enforced in critical

software, PLTs are ideal when considering gatekeeping and dispatching cross-module calls.

Combining all the above observations, we bridge this gap by proposing PLATYPUS, a defense mechanism that substantially reinforces the forward-edge (e.g., indirect calls and jumps) protections enforced by schemes such as CET and BTI. We demonstrate the efficacy of our approach through an implementation on top of CET for Linux x86_64 programs. Within our framework, indirect call and jump instructions cannot reach addresses outside of the current module, whether it is the main binary or a DSO, thereby rendering it infeasible for attackers to exploit arbitrary API functions. To achieve this, PLATYPUS introduces *execution jails* during program execution, imposing a strict separation of DSOs in virtual memory, enforced by an efficient address masking mechanism applied to all relevant instructions. Cross-DSO transitions are permitted exclusively via PLTs. Furthermore, to mitigate potential misuse of PLTs within a module, we separate those that are valid *indirect* targets from those that are only intended to be called *directly*. As we show, nearly all PLT entries belong to the latter category, which further prevents external functions from being accessible through indirect transfers.

PLATYPUS constitutes a lightweight mitigation applicable to both applications and libraries. It does not necessitate CFG construction or extensive static analysis, but instead leverages the Intermediate Representation provided by the LLVM toolchain for instruction instrumentation. Moreover, it preserves DSO sharing semantics and supports the coexistence of protected and unprotected modules, hence facilitating incremental deployment. Notably, our approach robustly handles edge cases involving indirect transitions, such as *callback* functions and dynamically loaded modules, ensuring completeness and applicability across diverse scenarios.

This is further supported by a comprehensive evaluation on 19 applications and 16 libraries, including glibc, encompassing binaries of varied complexity in both categories. We compared PLATYPUS to CET and found that, on average, PLATYPUS reduces the set of indirectly reachable external functions per module by more than *98%*, destroying the reachability of both known and novel exploitation dispatcher functions as well as function-level code reuse gadgets. Finally, we assessed performance in several complex and widely used programs, including Redis, SQLite, and Nginx, as well as on SPEC CPU2017. Our results show that the runtime overhead remains negligible, consistently below *0.5%* for real-world applications and below 0.72% on SPEC, thus demonstrating the efficiency of our defense. PLATYPUS is open-source, available at *https://github.com/Platypus2025/Platypus*.

2. Background and Motivation

2.1. Code-Reuse Attacks

Code-reuse attacks (CRA) are among the most prevalent techniques in program exploitation. In CRAs, once an attacker has successfully hijacked the control flow, typically by exploiting a vulnerability, they utilize existing code fragments, known as gadgets, found within the main program or its associated libraries. This attack paradigm emerged as a response to the introduction of the Data Execution Protection (DEP) mitigation, which prevents the execution of injected code, such as shellcode, on the stack. Different classes of CRAs can be distinguished based on the structure of the gadgets they utilize. Examples include Return-Oriented Programming (ROP) [65], Jump-Oriented Programming (JOP) [17], and Signal-based ROP (SROP) [18].

2.2. Control Flow Integrity

Control Flow Integrity (CFI) [7], initially proposed by Abadi et al. in 2005, is acknowledged as one of the most compelling ways to mitigate CRAs. Its main goal is to restrict the control flow paths that could appear during a program's execution, based (typically) on a pre-computed CFG. Given this CFG, each indirect control flow instruction is restricted to target only a specific set of valid pointers, limiting the reachability of possible gadgets that can be leveraged by an attacker. These targets may correspond to either backward edges or forward edges in the program's control flow. Backward edges consist of function return addresses, which are stored on the stack. Forward edges, on the other hand, are associated with indirect control-flow transfer instructions such as calls and jumps, whose targets are computed at runtime. A CFI scheme's security guarantees depend on its completeness and the size of its allowed target sets, as determined by the enforced CFG.

2.3. Intel CET

Despite the plethora of proposed CFI schemes, Intel's Control Flow Enforcement Technology (CET) [40] has emerged as the most prominent example deployed in practice on the `x86_64` architecture, particularly within the Linux ecosystem. Intel CPUs supporting CET entered the market in 2020, and the vast majority of Intel CPU-based systems now provide virtual support for it. The Linux kernel, along with popular distributions such as Ubuntu [68], enable CET by default. This establishes a paradigm in which CET will be present on all future Intel cores, thereby shaping the security landscape in which applications will operate.

CET introduces two hardware-backed key security mitigations: Shadow Stack and Indirect Branch Tracking (IBT). Shadow Stack robustly protects backward-edge pointers, such as return addresses, thereby effectively preventing CRAs like ROP. In contrast, IBT secures *forward-edges* within the CFG. Specifically, IBT restricts indirect control-flow transfers to designated entry points marked by the `endbr64` instruction, the vast majority of which correspond to function entries. As a result, under CET, the granularity of exploitable gadgets is elevated to the function level, preventing attackers from arbitrarily redirecting control flow to locations within a function. Although this does not, by itself, hinder the invocation of arbitrary functions, it raises the bar

for exploitation by complicating the process of setting up arguments, an essential step in almost every attack scenario.

2.3.1. Attacks Under CET. Unfortunately, the security guarantees provided by IBT are not particularly robust. Prior work [36], [44], [47] shows that exposure of whole functions as reusable gadgets still permits exploitation using Function-Oriented Programming (FOP). FOP-style attacks consistently rely on a common pattern: the use of an indirect call dispatcher embedded within a loop that iterates over an array of function pointers. By gaining control of relevant memory and redirecting execution to the dispatcher gadget, attackers can call arbitrary functions within the loop, chaining desired functionalities to set arguments and ultimately compromise the process. Consequently, while composing entire functions into FOP chains is generally challenging, the presence of such dispatcher gadgets can significantly facilitate, and in many cases prove essential for successfully bypassing CET.

2.4. Dynamic Shared Objects

Dynamic Shared Objects (DSOs) are binary libraries loaded at runtime to provide functionality that applications access through well-defined APIs. This design separates common or low-level functionality from application code, promoting modularity and simplifying maintenance. A key advantage of DSOs is their *shared* nature: once loaded, the same library image can be mapped by multiple processes simultaneously, reducing memory footprint and improving system efficiency.

Unfortunately, DSOs often suffer from *bloat*, as applications typically use only a fraction of their exported routines while the entire library remains mapped into the process' address space. This unnecessary code greatly expands the pool of available gadgets useful for CRAs, thereby enlarging the overall attack surface. Among all DSOs, glibc is the most prominent and security-critical, as it is loaded by nearly all Linux applications and exposes numerous system call wrappers and sensitive primitives. Consequently, it offers attackers abundant gadgets and privileged operations, making it a frequent and high-value target during exploitation.

2.5. Procedure Linkage Table (PLT)

In a typical scenario, an application or library (DSO) imports functions exported by other shared libraries via header files. This process creates explicit dependencies on the external DSOs that provide these functions. After compilation and linking, these dependencies are reflected in the binary's metadata, where the externally required functions are represented as imported symbols.

In Linux, the system manages these symbols through relocations, which populate the *.rel.plt* section of the ELF file. During linking, the linker emits special code stubs for these relocations and places them in the PLT, i.e., the *.plt* section. These stubs perform indirect jumps to the addresses stored in their corresponding entries within the *.got.plt* section, which forms part of the Global Offset Table (GOT). At runtime, the loader populates these GOT entries during symbol resolution so that each one points to the (virtual) address of the appropriate imported function. The necessity for this mechanism arises from the fact that the addresses of imported functions are not known during compilation, since libraries are built as Position Independent Code (PIC). Concurrently, with the Full RELRO mitigation in place, the contents of the GOT are read-only and cannot be overwritten, thereby rendering dispatch through the PLT both secure and reliable.

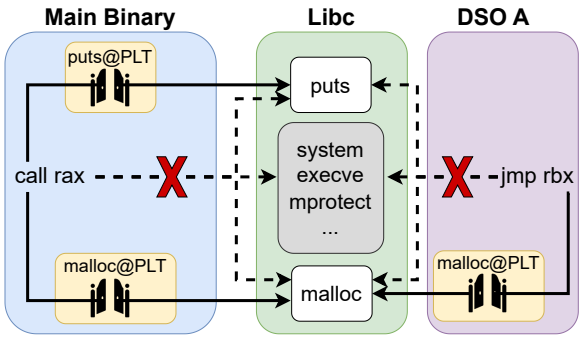

Figure 1. Module separation under PLATYPUS. Dashed arrows indicate infeasible indirect transitions.

At times, function symbols may appear in the *.rel.dyn* section, which is reserved for relocations unrelated with the PLT. This generally occurs for *address-taken* function symbols. In simple scenarios, compiler optimizations can determine when such symbols, typically stored in a variable, are invoked, and thus the linker generates a PLT stub for them. However, this is not always the case, particularly in libraries and applications with sufficiently complex code bases. As a result, when necessary, these symbols are accessed via indirect calls through the corresponding relocation, in contrast to PLT stubs, for which the compiler emits direct calls. We will revisit this distinction later in § 3.3.2.

2.5.1. PLTs under CET. Under CET, indirect jumps in PLT stubs must target `endbr64` pads. To accommodate this, IBT-compliant linkers emit two tables in the *.plt.sec* and *.plt* section respectively: the first includes an `endbr64` instruction followed by an indirect jump through the GOT entry, while the second places an `endbr64` prior to the code responsible for lazy symbol resolution, as in the non-CET case.

2.6. Motivation

Given the concept of PLTs, it appears counterintuitive from a security perspective to permit *arbitrary* indirect forward edges to target addresses within external DSOs. After all, PLT stubs were specifically designed to facilitate secure and efficient cross-DSO control-flow transfers, and their enforcement, especially under Full RELRO, is both straightforward and robust, effectively preventing CRAs from manipulating these transitions as can easily occur with indirect

code pointers. **Why, then, should cross-DSO control-flow transfers be allowed outside of PLTs, thereby enabling attackers to reuse arbitrary functions in other DSOs?** The current lax handling of cross-DSO transitions is particularly concerning, given that libraries, mainly glibc, expose a substantial number of sensitive functions (e.g., *system*, functions executing *syscalls*) that, if invoked, can result in complete process compromise. CET, steadily becoming the standard protection mechanism for binaries on Intel architectures, unfortunately remains vulnerable to this issue.

The preceding question constitutes the core motivation for the development of PLATYPUS. Our key novelty lies in the combined use of PLTs, address masking, and Intel CET to determine and enforce the validity of cross-DSO transitions. In particular, PLATYPUS leverages PLTs in a previously underexplored manner, transforming them from a purely linkage-oriented construct into a security-relevant enforcement mechanism. To achieve this, all indirect cross-DSO control-flow transfers are mediated through the associated PLT stubs. Accordingly it is impossible for any indirect call or jump within a DSO to target an external symbol for which a corresponding entry does not exist in its PLT. As illustrated in Figure 1, transfers denoted by dashed arrows, permitted by CET, are infeasible under PLATYPUS; cross-DSO transitions are restricted to their respective PLT entries.

2.7. Our Goals

Guided by the aforementioned motivation, we establish the following design goals that PLATYPUS should fulfill, with practicality as an essential consideration. We distinguish between equally-important security goals (SG) and auxiliary goals (AG).

- **SG1**: Limit cross-DSO control-flow transitions to a strictly defined set of authorized targets. In particular, we want to confine authorized cross-DSO targets per transition-originating DSO and not global for a whole process. Our main concern is to make it infeasible for an attacker to use arbitrary functions from shared libraries, except the ones expected.
- **SG2**: Provide comprehensive support for both the main executables and their associated shared libraries. We also want to ensure that critical libraries despite their complexity and special features (e.g., glibc) are protected as well.
- **SG3**: Provide robust security guarantees even during the gradual deployment of our scheme. We aim to prevent attackers from reaching arbitrary functions in other DSOs, even if those remain unprotected.
- **AG1**: Impose only negligible run-time and compile-time performance overhead. For this, heavy instrumentation (e.g., Intel Pin) and excessive static analysis (e.g., CFG construction) or extra requirements (e.g., link time optimization) should be avoided.
- **AG2**: Avoid library recompilation for each and every main program linking them. In other words, each shared library should be compiled once, irrespective of how many targets link against it.
- **AG3**: Retain library sharing. When multiple processes use the same shared library, only one copy of its code should be loaded into memory.
- **AG4**: Ease of deployability, thus facilitating straightforward and practical adoption.

3. Methodology

3.1. Threat Model

We assume an x86_64 platform equipped with a CPU that supports Intel CET. As discussed in § 2.3, under this scenario, the granularity of exploitable gadgets is restricted to those beginning with the `endbr64` instruction, thereby limiting permissible targets for indirect calls and jumps primarily to function entry points.

PLATYPUS aims to protect user-space processes on Linux systems. We presume that the OS enforces DEP (non-executable stack/heap) and the Full RELRO (no GOT manipulation) mitigation.

We assume that an attacker can exploit memory corruption vulnerabilities (e.g., arbitrary reads or writes) within the program, thereby diverting control flow. In this manner, the attacker can hijack function pointers or otherwise redirect function calls. No restrictions are placed on the number of times or the specific circumstances under which the attacker can trigger these vulnerabilities. Other software hardening techniques, such as ASLR, CFI schemes or defenses against data-only attacks, are orthogonal to our approach. We neither require nor exclude them. This threat model is consistent with prior work on CFI and software debloating.

3.2. Design Overview

PLATYPUS' primary goal is to enforce specific cross-DSO *"communication paths"* and, in particular, to restrict them to always leverage a PLT stub. **A DSO should reach an external function if and only if it possesses a PLT stub for this external function.** This highlights that PLATYPUS enforces restrictions at a per-DSO granularity for indirect cross-DSO call targets, as opposed to merely allowlisting accessible functions globally, as done in previous work [75].

The general design can be seen in Figure 1. As depicted, every single module's execution is restricted inside its own dedicated memory region within the process' virtual address space. External functions can only be reached through well-defined interfaces, in our case the PLT stubs. Every module has its own set of PLT stubs and therefore can access only the corresponding specific cross-DSO functions. For example, in Figure 1, although the function *puts* is required during the execution of the main binary, it cannot be accessed by DSO A. This restriction holds because only Main Binary contains the relevant PLT entry. In a similar vein, Libc can access no external code, since it contains no PLT stubs.

To achieve the memory separation we modify glibc's loader and employ a simple, yet highly effective and efficient masking technique, similar to those used in prior Software

```
; Original          ; Modified
...                 or      rax, ormask
...                 and     rax, andmask
call    rax         call    rax
```

Listing 1. Indirect calls before and after our instrumentation.

Fault Isolation (SFI) schemes [72], [48], [56], [67], by instrumenting indirect calls and jumps through a compiler pass. This masking technique mandates that an indirect pointer inside a module can only target addresses within the module itself. Furthermore, we modify the linker to emit all required PLTs (see § 3.3.2). We use LLVM's Clang compiler and LLVM's LLD linker.

The binary creation process is performed once per DSO. Subsequently, any program dependent on this DSO can utilize the mitigated version produced by PLATYPUS without further compilation or modification of the library (obj. AG2). Moreover, all programs on the same system can still share the memory pages of this mitigated instance (obj. AG3).

3.3. Execution Jails

Typically, indirect control flow transitions occur through function pointers. These pointers can be global variables, fields in a struct, non-PLT relocations for symbols (*.rel.dyn* pointers), or callback functions, among others. Either way, indirect control flow transfers can target both internal and external addresses with respect to the call-originating module (DSO or main binary). We refer to the first case as *intra-module calls* and the latter as *inter-module calls*.

Without our scheme, attackers can transform any intra-module transfer to an inter-module one, vastly expanding the set of gadgets they can exploit. Instead, under PLATYPUS, *every indirect control flow is intra-module*, with the sole exception of indirect jump instructions within the PLT stubs. As a result, inter-module transitions become unfeasible, which significantly limits the reachability of external API functions to *only those that the call-originating module actually requires*, as determined by its source code.

In the following, we describe how PLATYPUS restricts indirect calls. To this end, in § 3.3.1, we introduce *execution jails*: module-specific memory regions from which execution may only exit via dedicated interfaces. A lightweight compile-time instrumentation enforces that during execution a module cannot escape its jail. § 3.3.2 then describes how modules perform inter-module calls via well-defined PLT interfaces. Finally, in § 3.3.3, we outline how we can further distinguish between directly and indirectly-called PLT entries to further shrink the set of callable functions.

3.3.1. Separating and Enforcing Memory Bounds. The standard Linux loader already places modules in separate memory regions during startup. However, this alone offers no isolation guarantees, as modules can freely transfer control to other modules. Moreover, even with Address Space Layout Randomization (ASLR), in most cases, DSOs are loaded consecutive to each other [16]. To prepare for an effective and efficient module isolation, we separate each module in a dedicated *execution jail*. Each such jail is a memory region with a *unique* address prefix that is not shared by any other loaded module. In our prototype, each module is mapped to a 28-bit (256 MiB) memory range with a constant, dedicated 36-bit address prefix.

Having fixed-length region prefixes allows to efficiently enforce that, during execution, a module stays within its jail. To accomplish this, we mask all indirect control flow targets with two bitmasks, namely the *ormask* and the *andmask*. We define ormask and andmask as the start and end of the jail's memory region, respectively. Given the fixed address prefix, ormask is essentially the prefix. The andmask is the prefix padded with a zero bit and 27 high bits (0x7fffffff, i.e., 128 MiB). The 28th least-significant bit distinguishes code reachable indirectly (bit 0, part of andmask) from code reachable only through direct calls (bit 1, *not* part of andmask), as per § 3.3.3. Page-table permissions to refine read, write and execute privileges still apply as usual. This masking technique is similar to how SFI schemes enforce strict memory boundaries for untrusted components within a process; however, we leverage it to reinforce the forward-edge protection provided by schemes such as CET.

Listing 1 illustrates how we sequentially use these masks for every indirect function pointer in the code during a compiler pass. The `or` operation with the ormask ensures that the pointer references an address at least as large as the ormask, i.e., a lower bound of the targeted memory. A subsequent bitwise `and` with the andmask sets an upper bound to the reachable region. Consequently, the final pointer will always reference an address within the range $[ormask, andmask]$. Thus, even if an attacker modified the value of the register used in the indirect call, the final value utilized at runtime would necessarily fall within the previously established mask range. Since each DSO possesses its own distinct range, this ultimately means that *hijacked code pointers cannot target inter-module addresses*. We emphasize that our system operates under CET, which prevents attackers from jumping directly to call instructions to bypass the enforced masking. Additionally, CET ensures that if a masked, hijacked pointer does not point to an `endbr64` instruction, a Control-Protection exception is raised, terminating the process with a segmentation fault.

The two masks are emitted as GOT Relocations (*.rel.dyn* section) in the corresponding module and are populated at runtime by the loader. We have modified the latter accordingly to support these new relocation types.

The described masking mechanism is performant, introducing only two low-overhead instructions before each call and jump. Furthermore, the required static analysis is minimal (addressing objective AG1), as it primarily involves identifying and instrumenting indirect control-flow transitions in the LLVM Intermediate Representation (IR). Since IR code generation is a standard intermediate phase during compilation with the Clang compiler, our approach simply inserts the necessary instrumentation without altering this process. No additional instrumentation or CFG generation is required.

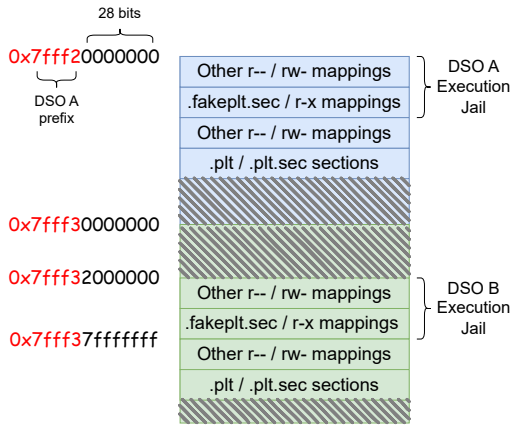

Figure 2. Separation of *.plt* and *.plt.sec* sections from execution jails in two example DSOs *A* and *B*. Gray lines represent unmapped addresses.

3.3.2. Support for Inter-Module Transfers. The aforementioned mechanism forbids any *indirect* inter-module call. However, compilers typically arrange desired inter-module calls through PLT stubs, which are then called *directly*. As PLATYPUS does not alter direct calls/jumps, the vast majority of inter-module transfers work seamlessly.

Nonetheless, sometimes imported (external) function symbols are not assigned a PLT stub by the linker but only a GOT Relocation inside the *.rel.dyn* section (cf. § 2.5). We need to take care of such symbols as the target address lies in a different DSO and therefore is outside of this module's execution jail. To overcome this problem, we modified the linker to emit a PLT stub for every external function symbol present in the *.rel.dyn* section. We call these stubs *fake PLTs* and place them in a dedicated section named *.fakeplt.sec*. We then adjust the associated relocations in the *.rel.dyn* section such that they point to the generated PLT stub rather than directly to the external function symbol. This changes the target of these relocations during indirect control flow transitions. Instead of pointing to the other module, they now point to an intra-module PLT, thereby constituting a valid intra-module call that falls within the module's jail.

3.3.3. Separation of Normal and Fake PLTs. An important distinction between fake and normal PLTs is that fake PLTs are always accessed via indirect code pointers due to their GOT relocations. Conversely, normal PLTs are not accessed indirectly (cf. § 3.3.2). We leverage these insights to further restrict the set of external functions accessible via indirect calls/jumps within a module. This restriction significantly increases the security of PLATYPUS compared to prior DSO isolation approaches [55], which still allowed indirect reachability of *all* imported function symbols.

In our scheme thus far, the only way to reach an external API function through an indirect pointer—whether hijacked or not—is via a redirection to either a fake or a normal PLT stub. Raw pointers that might initially target addresses external to the current DSO, after undergoing the enforced masking, ultimately point to locations within the current DSO. However, since normal PLT stubs are never legitimately accessed through indirect control flow transfers, we can segregate them into a separate memory mapping, the addresses of which do not fall within the DSO's $[ormask, andmask]$ range. This separation significantly narrows down the set of cross-DSO functions that are reachable via Intra-Module calls to only those for which the respective DSO provides fake PLTs. For example, even if a module invokes sensitive functions (e.g., *system*), attackers cannot leverage their PLT stubs via indirect code pointers unless these functions are *address-taken* within the module.

To isolate the normal PLT stubs to addresses outside the masking range, we create a separate executable mapping that contains only the *.plt* and *.plt.sec* sections and place them *outside of the module's jail*. This mapping is distinguished by having the 28th least-significant bit set (as described in § 3.3.1). All remaining executable sections (e.g., *.text*, *.fakeplt.sec*, *.init*, *.fini*, etc.) remain contiguous in memory within the module's jail, where the 28th bit is *not* set. Since the *.plt* and *.plt.sec* section pages are located at higher addresses than the andmask, they remain out of reach for indirect calls. For space optimizations and to eliminate gaps between mappings, each DSO is mapped so that its andmask aligns with the highest address of the executable mappings whose 28th least-significant bit is not set. For this to hold, the least significant 27 bits of the address need to be of type $2^n - 1$ for some natural number n, otherwise the andmask could exclude addresses from the range $[ormask, andmask]$. As a result, a DSO may not be mapped exactly at an address with the last 28 bits zeroed but gets shifted to a higher one, which is always page aligned. The above are presented in Figure 2. The same figure highlights the unique prefixes of each DSO (red part of the addresses), as introduced in § 3.3.1. Observe that (under our optimization) if DSO B was mapped at address `0x7fff30000000`, the resulting andmask would be `0x7fff35ffffff`. This would not support certain pointers within the range $[$`0x7fff30000000`$,$ `0x7fff35ffffff`$]$, such as `0x7fff340000000`, as these pointers would be modified when the `and` instruction (with the andmask) is applied.

3.4. Support for Callbacks

The aforementioned scheme already covers the vast majority of scenarios in which a DSO requires access to an external function residing in a different module. Nonetheless, shared libraries and large-scale programs may use inter-module calls targeting pointers for which neither normal nor fake PLTs were generated. In other words, cross-DSO calls may be made to functions that are not known at compilation or linking time, and thus lack corresponding relocations. Consequently, we need to extend our scheme to support such cases as well. In this section, we focus on callback pointers (CPTR), which fall into this category. We defer the discussion of dynamically-loaded modules (§ 3.5) and other corner cases (§ A).

3.4.1. CBG Annotation. In general, CPTRs are function pointers which are passed as arguments to functions. We denote these functions receiving callback pointers as callback getters (CBG). Some of these CPTRs are invoked directly by the CBGs, while others may be stored in variables and data structures, such as structs, and invoked at a later time. Either way, masking the pointers destroys the expected control flow of the program if the caller and target of the CPTR reside in different DSOs. Consequently, we need to extend our scheme to support callbacks.

To automate this process, PLATYPUS examines each function's signature and annotates CBGs as such. This automated annotation is crucial, as it greatly aids identification of CBGs within the compiler passes employed during symbol gathering (see § 3.4.2). We highlight that we do take care of *typedefs* in the datatypes and other complex cases like *macros* during annotation. This step constitutes the only addition or modification we make to the files containing the source code. PLATYPUS identifies the vast majority of CBGs in our extensive test suite of programs and libraries. Nevertheless, full coverage cannot be guaranteed due to the diversity of CPTR datatypes (e.g., *void* *). Such rare cases can, however, be readily supplemented by developers.

3.4.2. Symbol Gathering. Next, we collect the functions that each DSO may need to call and which fall under the corner cases we discussed. For this PLATYPUS incorporates two separate LLVM compiler passes that record the relevant information, while keeping the static analysis of the underlying IR minimal. In these passes we mainly leverage debug information, for more precise and complete results. The logged information consists of the following:

CPTRs. We record instances where function symbols are passed as arguments to CBGs, either directly or via global function pointer variables. For local variables used as CPTRs, our analysis considers only whether a function symbol is stored in them within the same function scope. This approach proved complete even for the largest real-world programs in our evaluation, while keeping the required static analysis negligible. In every case, we record the function names of both the CPTR and the CBG.

It is important to note that, since the symbol names of recorded CPTRs are known at compile and link time, any external function symbol referenced by the current module being instrumented will have a corresponding fake PLT stub, as it is *address-taken*. Accordingly, we can filter out those cases where the CBG symbol is internal to the module being instrumented by PLATYPUS. If an internal function is intended to call the CPTRs, fake PLT stubs are available for use, and the previously introduced masking mechanism already supports such scenarios. Therefore, the essential information to retain, concerns cases in which the CBG is an external symbol imported from another DSO (*A*), and the CPTR (*cptr_to_A*) is a symbol defined in a module other than A. Simply put, if the current DSO is ever used, then the DSO A should somehow be ready to support calling indirectly a pointer targeting *cptr_to_A*. For each instrumented DSO, we thus create a dictionary mapping receptor DSOs (e.g., A in the previous example) to lists of their corresponding CPTR symbols. Listing 2 illustrates an example. Calls to CPTR arguments are instrumented differently, as described in § 3.4.3.

Struct datatypes. Structs often embed function pointers in their fields, which are later invoked indirectly. Additionally, many DSOs export functions that accept, as arguments, struct datatypes (or pointers to them) containing function pointers within their fields. Since these functions are exported, it is safe to assume that a different module, which imports them, may invoke them while passing as an argument its own struct populated with arbitrary function pointers. Such cases are frequently encountered inside libraries, especially in cryptography-related ones (e.g., *libcrypto* and *libssl*). Having all these in mind, we need to record different types of information related with struct datatypes.

For each DSO (e.g., DSO B), we log struct datatypes that contain function pointers in their fields and are used as arguments to exported functions. Whenever, inside B, a call is invoked through a function pointer belonging to such structs, we instrument it differently, as the pointer may have been populated by an external module (see § 3.4.3). To complete this process, in every other DSO that depends on B and utilizes B's struct datatypes, we log the function symbols being assigned to them, while also instrumenting calls from their fields properly. These symbols could potentially be passed as arguments to functions imported from B. We record this information as a dictionary, mapping imported DSOs to the corresponding symbols that need to be anticipated by them, as was done in the CPTR case.

One aspect we also log in each DSO is struct datatypes in which CPTRs are stored. For this, we leverage the fact that the CBGs are annotated, enabling us to easily identify them and determine whether, inside a CBG, the CPTR is stored within a struct. If this is the case, we record the struct datatype in order to instrument it differently (see § 3.4.3).

On all the above we do not rely on excessive or time consuming static analysis, but rather leverage the capabilities of LLVM's IR plus the debug information.

```
{
  "DSO A": ["cptr_to_A", "cptr2_to_A"],
  "DSO B": ["cptr_to_B"]
}
```

Listing 2. Dictionary of DSO C dependent on DSO A and DSO B

3.4.3. Callback-Aware Instrumentation. When calling CPTRs or function pointers from specific struct datatypes (cf. § 3.4.2), we adapt the instrumentation to account for these cases as per Figure 3. We leverage the `r11` register to store the initial pointer to be called, since it is a temporary register that can be clobbered between calls. If the pointer corresponds to an Intra-Module call, the masking mechanism does not alter any of its bits. In this case, we simply execute the call (line 11) and continue. In contrast, if the

masks alter the pointer, the target is not an intra-module call but an exploitation attempt or a valid corner case.

In such cases, each DSO imports a uniquely named *DSO_callback_table* symbol corresponding to a PLT stub in the module. This stub redirects control to a special function *linked with the main binary*, whose role is to check whether the pointer in `r11` exists among the DSO's anticipated symbol list. This list contains all callback symbols collected for the DSO as described in § 3.4.2. If `r11` matches an entry, control jumps to the corresponding target. Otherwise, the program terminates with a `hlt` instruction, since the pointer is deemed unexpected. Thus, even if an attacker hijacks such pointers, they can only redirect them to internal or listed callback functions of the DSO.

We link these special functions and symbol lists with the main binary because different programs have different library dependencies. This means the callback lists are unique per DSO and per program. Our method enables modules to be compiled once and reused across binaries, without modifying shared library code and while still handling all corner cases. Functions in the callback tables are represented as relocations in the main binary, each pointing to the DSO base address plus the function's offset. Direct symbol relocations are insufficient, since non-exported functions are often used as callbacks and cannot be resolved by the loader.

As stated above, each DSO maintains its own callback table. Notably, there are cases, mainly involving initialization and finalization routines, where the exact set of pointers called by certain CBGs or struct datatypes can be identified after the process described in § 3.4.2. In these scenarios, a dedicated callback table can be assigned to the relevant CBGs or datatypes, and the indirect transitions can be replaced with direct calls to the PLT corresponding to this table, ensuring that such transfers can only reach their intended targets. This approach is revisited in § 4.2.2 to demonstrate how PLATYPUS prevents FOP attacks like the ones mentioned in § 2.3.1.

3.5. Support for Dynamic Loading

Programs may load a DSO at runtime using *dlopen* and invoke its functions by dynamically retrieving pointers to them via *dlsym*. These pointers are then called indirectly, and their addresses do not reside within the execution jail of the invoking DSO. Furthermore, they are typically not included in the corresponding callback table introduced above. Thus, these cases require explicit handling.

Generally, application-loaded libraries are particularly challenging to manage. Despite intensive static analysis, many CFI and debloating schemes still encounter difficulties in supporting them. Either loading such libraries leads to program crashes or the schemes' security guarantees are substantially undermined. Schemes that do support these cases employ specialized runtime handling, such as on-the-fly CFG and call target recalculation [13] or even rewriting code pages [32]. However, such approaches incur performance overhead and must additionally safeguard the brief

```
1   ; Original:
2   call    rax
3   ...
4
5   ; Modified for corner cases:
6   mov     r11, rax
7   or      rax, [rip+ormask_offset]  #ormask
8   and     rax, [rip+andmask_offset] #andmask
9   cmp     r11, rax
10  jne     call_target
11  call    rax
12  jmp     after_call
13
14  call_target:
15  call    DSO_callback_table
16  after_call:
17  ...
```

Figure 3. Extended instrumentation for corner cases

intervals during which memory regions critical to CFG enforcement remain writable, thereby potentially introducing attack vectors. Moreover, the strategy of rewriting code pages, particularly within libraries, directly violates our objective AG3.

Instead, we leverage the observation that calls to *dlopen* and *dlsym* typically exhibit predictable patterns that can be profiled using standard workloads and test suites. Prior work has made similar observations [8], [60]. We hence use profiling to determine which modules are *dlopen*ed and the arguments passed to *dlsym*. With this, we preload the necessary DSOs on startup and generate the appropriate PLTs for *dlsym*ed symbols in each module. Especially for DSOs, to avoid coupling them to individual binaries which would violate objective AG2, these additional PLTs serve as placeholders rather than being bound to specific symbols. Their binding is deferred to runtime, with the main binary supplying the necessary information via special exports.

Note that the majority of shared libraries do not directly invoke *dlopen*. A prominent practical example is glibc, which occasionally loads Name Service Switch (NSS) libraries to perform system-level lookups involving hostnames, usernames, and related information. Despite their relative rarity and the extra complexity they introduce, PLATYPUS is capable of effectively handling dynamic loading cases, including those involving NSS libraries.

4. Evaluation

We evaluate PLATYPUS based on three criteria: correctness (§ 4.1), defined as the absence of failures or crashes during benign execution; security (§ 4.2), assessed by the reduction in cross-DSO indirect control-flow transfers and FOP dispatcher gadgets; and performance (§ 4.3), measured in terms of runtime overhead and binary size changes.

All experiments were conducted on a general-purpose Ubuntu (version 24.04.1) host equipped with 8 Intel Core i7-9700K CPUs. Each benchmark and its required DSOs were instrumented using our LLVM pass, and the same compilation (`-O3,-fcf-protection=full,-g`) and link-

Table 1. BENCHMARK SUITE. EVERY BENCHMARK IS FOLLOWED BY ITS VERSION NUMBER.

rm v9.7	mkdir v9.7	gzip v1.14	make v4.4.1	Nginx v1.28.0
date v9.7	sort v9.7	grep v3.12	lighttpd v1.4.79	Redis v8.2.0
uniq v9.7	tar v1.35	objdump v2.44	wget v1.25.0	SQLite v3.50.4
chown v9.7	bzip2 v1.0.8	bftpd v6.3	memcached v1.6.38	

ing (`-Wl,-z,relro,-z,now`) flags were applied. DSOs were built as position independent executables (`-fPIC`). We employed our modified LLVM (version 20.1.0) clang compiler and lld linker (`-fuse-ld=lld`) for compilation and linking, respectively. For fair comparison, all involved programs and libraries were also compiled with an unmodified compiler and linker of the same version. Notably, and unlike most related work, our suite includes glibc (v2.41), compiled with Clang [74].

To rigorously evaluate our defense against the three criteria outlined previously, we assembled a comprehensive suite of programs. We placed exclusive emphasis on real-world applications that 1) are widely deployed, 2) span a broad range of complexity, and 3) require, during their execution, anywhere from a few to many shared libraries. Our final selection, depicted in Table 1, encompasses 19 diverse applications, ranging from essential GNU Coreutils binaries to complex network applications and servers (lighttpd, bftpd, wget, Nginx), as well as widely adopted data management systems such as Redis and SQLite. Collectively, these programs require 16 distinct libraries of varying complexity. Table 6 further confirms the suitability of our benchmarks: most applications exercise a substantial number of DSOs (second column), while in some cases a subset of these libraries is also loaded dynamically (third column). By utilizing such a substantial and diverse set of libraries, our experimental setup closely mirrors the challenges found in realistic deployment environments, and enables a thorough assessment of our mitigation's impact, which specifically targets indirect cross-DSO transitions. In addition, we also evaluated PLATYPUS on SPEC CPU2017 [21] to allow comparison with previous and future approaches. Importantly, our benchmarks remain consistent with those used in prior work [8], [32], [19], [13], [46].

4.1. Correctness

We aim to ensure that PLATYPUS instrumentation does not compromise the correctness of the benchmarks, by introducing crashes during benign execution. To verify this, we execute each benchmark using its provided test suite, which is typically bundled with the source code files. These test suites exercise numerous code paths that the compiled binary must be able to handle. As such, successful completion of these test suites provides strong evidence that a program remains functional.

We evaluated every benchmark for correctness. The only exception was *bftpd*, for which no test suite was provided with the program files. For this benchmark, we performed manual testing to examine its various functionalities. All programs compiled successfully without errors or the need for source code modifications. Tests passed successfully without faults as well for all the benchmarks. Notably, for the larger and more complex programs (e.g., Redis, SQLite, Nginx), the test suites comprised thousands of test cases, with SQLite alone executing more than 300,000 tests. For Nginx, we enabled support for *HTTPS (SSL/TLS)*, *HTTP/2*, *gzip* compression, real IP resolution, server status monitoring, and Layer 4 *(TCP/UDP)* proxying with *SSL/TLS*, thereby increasing both the number and coverage of executed tests under the Nginx test suite [29]. The above provide strong evidence that PLATYPUS preserves the correct behavior of both the programs and the DSOs to which it is applied. Moreover, the fact that this holds for both small and large binaries, encompassing various degrees of complexity and covering a plethora of scenarios in code, further reinforces the previous point.

Although some CFI schemes (e.g., [13], [32]) support these benchmarks, others such as Clang CFI [26] break two of them. Specifically, when applied to Nginx and Redis, Clang CFI causes these applications to crash when evaluated with their respective test suites. This is particularly noteworthy, as Clang CFI is the most widely deployed fine-grained CFI scheme in practice [15]. In contrast, PLATYPUS successfully supports such programs, thereby demonstrating its applicability in real-world scenarios.

4.2. Security Analysis

PLATYPUS transforms every inter-module transition into an intra-module one. Even when an attacker has successfully hijacked code pointers, the masking mechanism (see § 3.3.1) enforces that the final pointer called remains within the execution jail of the current module. This design makes it impossible for attackers to invoke arbitrary functions in external modules, restricting them solely to those for which fake PLTs exist, since normal PLTs are not part of the jail. The importance of this PLT separation becomes evident in Table 5, which demonstrates that fake PLTs constitute only a minority of the imported symbols in each module. Therefore, it becomes significantly more challenging to locate and chain useful gadgets (of function granularity), as such gadgets are typically not confined to a single module.

Table 2 further visualizes this restriction. Specifically, we demonstrate the reduction in valid indirect cross-DSO transitions (per module) when our scheme is employed, compared to CET. Recall that under CET, every `endbr64` instruction (in external modules) serves as a valid landing pad for indirect control flow transitions. The third column illustrates that the number of valid CET function targets is huge, in the ranges of thousands to tens of thousands. The fourth column shows the absolute number of reachable

cross-DSO targets per DSO when protected with PLATYPUS, including fake PLT stubs (in *.fakeplt.sec* section) plus the module's callback table entries. Using these numbers as baseline, we observe that the reduction is `98%` or more in every module across all cases (sixth column). Among the remaining few to a few hundreds of valid call targets under PLATYPUS, it is much more challenging to construct valid code-reuse gadget chains that cross DSOs.

Note that functions in the callback tables are accessible only from indirect call or jump instructions instrumented with the extended masking mechanism, and not from all such instructions (cf. § 3.4). The fifth column thus presents the number of functions in the callback table of each module. Their small number demonstrates that only a minor subset of functions serve as callbacks within the programs. We can thus safely conclude that cross-DSO transitions are strictly restricted to functions required during the runtime execution of the programs, and that functions which should never be called are not accessible from external modules.

Table 2. NUMBER OF INDIRECTLY ACCESSIBLE CROSS-DSO `ENBR64` PADS UNDER CET (COL. 3) AND PLATYPUS (COL. 4-6), FROM THE PERSPECTIVE OF THE RESPECTIVE MODULE. CT = CALLBACK TABLE.

	Module	CET	PLATYPUS	CT	Red. (%)
Redis	redis-server	3739	6	0	99.84
	libc.so	4961	52	52	98.95
SQLite	sqlite3	6873	43	0	99.37
	libreadline.so	7452	3	2	99.96
	libtinfo.so	7856	3	0	99.96
	libncurses.so	7347	3	0	99.96
	libm.so	7242	18	18	99.75
	libz.so	8064	0	0	100.00
	libc.so	4472	40	40	99.11
Nginx	nginx	17986	6	0	99.97
	libpcre2-8.so	20124	2	2	99.99
	libcrypto.so	9551	84	83	99.12
	libz.so	20085	6	6	99.97
	libssl.so	17182	31	14	99.82
	libcrypt.so	20171	21	0	99.90
	libc.so	16533	166	166	98.99

4.2.1. Libc protection. PLATYPUS does protect glibc (both in- and outgoing control transfers) despite its high complexity. This is critical, as libc is extremely rich in gadgets useful to attackers and, more significantly, it is the module responsible for executing syscalls. Consequently, any core functionality, like the ones that normally get leveraged by exploits, almost exclusively relies on libc functions. Our scheme restricts the arbitrary use of these functions at per-module granularity, permitting access only to those required by each module. For example, in Nginx, of the six cross-DSO `endbr64` pads accessible through indirect control-flow transfers, only one targets a libc symbol (*__libc_start_main_impl*), which is invoked exclusively during program startup. In Redis, all six such transitions point to libc functions (*__libc_start_main_impl*, *malloc*, *calloc*, *realloc*, *free*, and *strdup*). Crucially, these functions do not provide direct opportunities for process compromise. This is not coincidental, as sensitive functions (e.g., *system*, *execve*) are rarely *address-taken* within a module.

In contrast, most CFI schemes (e.g., [13], [32], [52]), including state-of-the-art approaches, have not been evaluated on glibc. This is primarily due to limitations in earlier Clang versions, which lacked support for compiling glibc. Consequently, these approaches adopt musl libc as a substitute. However, glibc is the standard C library for major Linux distributions such as Ubuntu, Debian, and Fedora, and many applications are explicitly developed with glibc in mind. While musl libc provides similar functionalities, notable limitations and compatibility issues persist [50]. Worse, some schemes do not even support musl libc, such as Clang CFI. As a result, since libc remains unprotected, Clang CFI permits any (!) indirect control-flow transfer to libc functions. Without proper integration of glibc, a CFI defense remains practically incomplete, an issue that PLATYPUS effectively addresses.

4.2.2. FOP Gadget Reduction. With PLATYPUS enforced, state-of-the-art FOP attacks described in § 2.3.1 are also mitigated. We emphasize that the most challenging aspect of such attacks lies in argument setup. For instance, while control flow can be hijacked to invoke *execve*, its arguments must also be precisely controlled. To demonstrate that PLATYPUS mitigates such FOP-style attacks in general, we search for FOP-enabling dispatcher gadgets in glibc. We focus on glibc for two primary reasons. First, glibc contains one of the richest sets of available gadgets among system libraries. Second, successful exploitaion typically requires invoking a syscall or a similarly privileged function, which almost always resides in libc. Under PLATYPUS, the reachability of such functions from other modules is substantially restricted.

Prior work [36], [44], [47] identified only one exploitable dispatcher gadget in glibc, as detailed in [47]. To extend this analysis, PLATYPUS incorporates a Ghidra script for exhaustively searching for potential dispatchers, which we applied to glibc (v2.41). The script identifies all indirect call instructions occurring within loops and excludes loops that would clobber registers required for dispatcher control. Among the potential dispatchers detected, only two can actually be used for FOP attacks: the previously known gadget and an additional candidate located within *__nptl_deallocate_tsd* function. A powerful attacker able to control memory and redirect a libc pointer to either of these functions could potentially mount a FOP-style attack.

These dispatcher gadgets are rare and share a notable commonality: association with initialization or finalization routines. For example, the gadget in [44] leverages *_initterm* function in *msvcrt.dll* that initializes pointers at C++ program startup. The gadget described in [47], located within glibc's *_dl_call_fini* function, is associated with exit routines and is responsible for invoking pointers registered in the *.fini_array* and *.fini* sections. Similarly, the additional gadget we identified, calls destructors registered by the user through APIs, which are executed when threads exit and their thread-local storage is deallocated. PLATYPUS effectively protects

against such scenarios by leveraging knowledge of the exact set of target symbols, eliminating indirect calls and permitting only direct calls to the corresponding callback table (see § 3.4.3). Notably, PLT stubs cannot exist for these two glibc dispatchers, as they reside in non-exported functions. More generally, under PLATYPUS, dispatchers and their dispatched functions can only be reached indirectly if the corresponding module that invokes them provides PLT entries for those targets, with enforcement applied at a per-module granularity. For example, the dispatcher described in [47] resides within the loader function *_dl_call_fini* and subsequently performs indirect transitions to functions in libc. PLATYPUS prevents such indirect cross-DSO transitions, as the loader does not provide the necessary PLT stubs for the invoked libc functions.

4.2.3. Intra-Module Hijacking. PLATYPUS is not a CFI scheme but a mitigation mechanism over CET that provides strict guarantees for indirect cross-DSO transitions, which play a central role in exploitation. For *intra*-DSO transitions PLATYPUS likewise reduces the attack surface, as normal PLT stubs are no longer reachable indirectly. The resulting security gain is significant, because the number of fake PLTs per module is small (see Table 5). Normal PLTs remain reachable, but only transitively: an attacker must first target a function that subsequently invokes these PLTs (directly). In most cases, however, mere reachability is insufficient; the attacker must also control the function's arguments, which is harder due to the required intermediate call. Hence, state-of-the-art CRAs must combine memory control with specific dispatcher gadgets, a demanding prerequisite that, as we have shown in § 4.2.2, PLATYPUS is designed to prevent. Moreover, in many practical cases, existing functions hardcode arguments to sensitive functions such as *execve*. For instance, *memcached* invokes *system()* with a static argument [3]. Similarly, the sensitive callback argument of *pthread_create()* is hardcoded in *nginx* [4], *memcached* [2] and *libselinux* [1].

PLATYPUS though has been designed with a strong threat model in mind, under which a powerful attacker could theoretically mount attacks by manipulating code pointers in memory, chaining indirect and direct calls to set arguments and invoke sensitive targets (e.g., *system*). Given PLATYPUS' strict enforcement of cross-DSO boundaries, the practical risk is largely confined to glibc, as it naturally concentrates the system's most security-critical functionality. From an attacker's standpoint, the only remaining useful code pointers lie in global function pointers inside libc (e.g., hooks) or, more generally, in function pointers stored in global objects such as arrays. These opportunities, however, are both limited in number and well-documented within the glibc community. Ongoing hardening efforts have further narrowed this attack surface, notably with the removal of legacy constructs like the *malloc* hooks [53] and the introduction of additional hardening checks [28].

Moreover, PLATYPUS' flexible design allows for even tighter control over such vectors. Libc functions that are not invoked indirectly during benign execution (e.g., *system*) can be isolated by the linker outside of libc's execution jail, as with normal PLTs. To ensure robustness, all functions that directly invoke, or are directly or indirectly invoked by these sensitive functions should also be isolated, relationships that can be identified through static analysis. Applying this approach to libc, we found that functions like *system* and *execve* can be effectively protected, making them inaccessible from within libc's execution jail. Importantly, this isolation also protects crucial functions such as *pthread_create* and *__clone3*. Hence, even in the theoretical attack scenarios described above, PLATYPUS further limits the attackers, making full process compromise substantially more difficult.

Table 3. OVERHEAD OF PLATYPUS (REAL-WORLD APPLICATIONS)

Application	Overhead	Application	Overhead
Bftpd	0.00%	Redis	0.39%
Memcached	0.22%	SQLite	0.28%
Nginx	0.44%	Lighttpd	0.00%
Nginx (ramfs)	0.47%		

4.3. Performance Evaluation

4.3.1. Runtime performance on Real-World Applications. To demonstrate the efficiency of our scheme, we measure its impact on the runtime performance of protected programs using complex, widely adopted benchmarks representative of demanding real-world scenarios. Our aim is to provide evidence that PLATYPUS constitutes a practical solution for protecting high-performance binaries. Accordingly, we assessed the runtime performance implications for a set of widely-used applications: Bftpd, SQLite, Memcached, Redis, Lighttpd, and Nginx. As a baseline, the original uninstrumented, but CET-protected, binaries are used. Every experiment was repeated 60 times to diminish small variances that sometimes appear between runs. We also pin each process to a dedicated core, disable CPU boost, and configure each CPU core to operate at a constant frequency (3.6 GHz) to minimize fluctuations during benchmarking. Our findings are depicted in Table 3.

For ***Bftpd*** we used *ftpbench*[63] to create a rich testsuite examining how the server behaves under a significant load of concurrent connections. This benchmark focused in operations like uploading and downloading files in the FTP server. The final degradation of throughput was 0%.

Regarding ***Redis***, we used *memtier_benchmark*[62] with two worker threads managing a total of 128 concurrent client connections. The workload had a 1:10 SET to GET ratio, with 32-byte data objects, and was executed for 60 seconds. To minimize I/O and network latency, both client and the redis-server ran on the same host. Using more threads in the *memtier_benchmark* would not increase load, as the redis process is single-threaded and was already saturated. We observed a negligible throughput degradation of 0.39%.

For ***Memcached***, we replicated the *Redis* experiment, except both *Memcached* and *memtier_benchmark* were con-

Table 4. OVERHEAD OF PLATYPUS (SPEC CPU2017)

Application	Overhead	Application	Overhead
600.perlbench	0.72%	602.gcc	0.43%
605.mcf	0.27%	625.x264	0.57%
657.xz	0.00%		

figured to use 4 threads, leveraging Memcached's multi-threading support. Since our machine has 8 cores, we pinned memtier's and Memcached's threads to separate cores to avoid contention and context switches between them, that could mask performance overhead. The number of simultaneous client connections (128) led once more to full CPU utilization. The throughput was again negligible (0.22%).

SQLite was evaluated using the *speedtest* utility, which benchmarks various database operation scenarios. To avoid I/O masking performance overhead, we used an in-memory database (*:memory:*). Since the default parameters resulted in very short runs (∼2.5 seconds), we increased the test size to 300 (*--scale 300*) to apply a heavier load. The protected binary exhibited a performance overhead of 0.28%.

Nginx and ***Lighttpd*** were evaluated using the *wrk* [33] tool, which stress-tests web servers. *Wrk* was configured with 4 threads collectively making 400 concurrent requests, with each test lasting 60 seconds. The requested file size was set to 20KB to avoid masking performance overhead with I/O, as it can happen with big files. To eliminate network latency, both servers and *wrk* ran on the same host, communicating through the loopback interface. Each server was configured to use 4 worker processes, saturating the CPU cores. *Lighttpd* exhibited 0% throughput degradation, while *Nginx* showed a 0.44% degradation. To further eliminate potential I/O overhead, the same experiment was repeated for Nginx with the file served from RAM using *ramfs*. The degradation remained negligible, specifically 0.47%.

4.3.2. Runtime performance on SPEC CPU2017. To facilitate comparison with prior approaches that typically evaluate using SPEC benchmarks, we further evaluated PLATYPUS on SPEC CPU2017. As before, we used the same experimental setup: each process was pinned to a dedicated core, CPU boost was disabled, and a constant frequency was enforced across all cores. The results are presented in Table 4. Our baseline consists of the original, uninstrumented yet CET-protected benchmarks from the SPECspeed 2017 Integer suite. We used the *ref* workload and executed each benchmark 10 times. Note that our prototype currently supports only C binaries; therefore, we evaluated only the C benchmarks from the suite. A discussion on extending PLATYPUS to support C++ applications is provided in § 5.

The results consistently show that the performance penalty incurred by PLATYPUS remains negligible in most cases and never exceeds 0.72%. Combined with the previously discussed results on real-world applications, this demonstrates that restricting cross-DSO indirect transitions using the techniques presented in § 3 not only improves security guarantees (as analyzed in the preceding security evaluation) but also remains performant and efficient.

```
endbr64
or      rcx, qword ptr[rip + ormask_offset]
and     rcx, qword ptr[rip + andmask_offset]
jmp     qword ptr[rip + qsort@got.plt]
nop
```

Listing 3. PLT stub of *qsort*. CPTR is on `rcx` based on System V ABI.

4.3.3. Binary Size Increase. We also evaluated the impact of PLATYPUS on binary size for both programs and libraries. As expected, the masking instrumentation and the linkage of the special functions supporting the callback table mechanism (see § 3.4.3) result in increased code size. The results are presented in Table 8 for the benchmarks and Table 7 for DSOs. For DSOs, our scheme introduces a small size increase, averaging 5.78%. Benchmarks exhibited larger average increases, with the highest percentages observed in programs smaller than 400KB.

5. Discussion

In this section we discuss some functionalities that PLATYPUS supports and also some of its limitations.

Arguments to CBGs. One corner case described in § 3.4 involves DSOs passing CPTRs to external CBGs, scenario which introduces a potential attack vector. According to the extended masking, intra-module transitions are permitted. Consequently, an attacker could hijack the CPTR argument and redirect the control flow to any arbitrary function internal to the DSO to which the CBG belongs.

To protect against such attacks, we instrument both the normal and fake PLT stubs corresponding to external CBGs in every module. Specifically, we apply the bitmasks to the argument, based on System V ABI, of the CBG that contains the CPTR. The rationale for this approach is that the symbols passed as CPTRs are either local or external address-taken symbols. In the latter case, they possess a fake PLT stub. Importantly, both cases fall within the execution jail, ensuring that the masking does not modify the CPTR. However, if the CPTR was hijacked to point to an arbitrary function within the DSO containing the CBG, the masking would modify it so that it refers to an address inside the current execution jail.

To achieve this, we modified the linker to increase the size of the PLT stubs in *.plt.sec* and *.fakeplt.sec* to 32 bytes. Currently, PLATYPUS supports only one argument check, but if a CBG accepts multiple CPTRs, additional checks can be integrated into the respective PLT. An example is presented in Listing 3 with the *qsort* function. Based on its prototype, the CPTR is passed on the fourth argument.

Modular Support. As discussed in § 6.1.2, current CFI schemes struggle to support uninstrumented modules. In contrast, PLATYPUS accommodates these modules without

crashes or loss of security. More specifically, while uninstrumented modules can call arbitrary pointers, other DSOs (or the main binary) can access unprotected functions only if they include a PLT stub or an entry in their callback table for those functions. Consequently, each module's access to unprotected code is inherently tightly limited by our scheme: even if third-party libraries contain vulnerabilities, their exploitable functions are only reachable by modules with legitimate access. In § A.4, we discuss further how PLATYPUS can even handle CPTRs in unprotected modules.

Multi-threading Support. PLATYPUS fully supports multi-threaded applications without compromising security or introducing crashes. Because both the *ormask* and *andmask* for each module are accessible to every thread through relocations, execution jails are, by design, thread-agnostic. Moreover, the use of the `r11` register during extended masking does not create conflicts, as each thread operates on its own set of registers independently.

C++ Support. Although PLATYPUS can instrument C++ code, our current symbol gathering passes (see § 3.4.2) do not account for language-specific features such as *classes*, *inheritance*, and *virtual tables*. We leave the extension of our logic to support these features to future work, as it is primarily an engineering effort to capture the relevant symbols in the IR produced during C++ code compilation.

6. Related Work

In this section, we survey related work in the areas of CFI, Debloating and Compartmentalization. We emphasize, however, that while PLATYPUS does not belong to any of these categories, it draws inspiration from all three.

6.1. Control Flow Integrity

As introduced in § 2.2, CFI's security guarantees depend on its completeness and on the size of the allowed target sets, as determined by the enforced CFG. Depending on the granularity of these sets, CFI schemes are typically classified as *coarse-grained* or *fine-grained*.

6.1.1. Coarse-grained CFI. Coarse-grained CFI schemes, while limiting exploitation paths during CRAs, often prioritize performance over comprehensive security guarantees. Inevitably, the CFG they compute is overly permissive, exposing critical functions located normally in shared libraries (e.g., libc) which are (almost always) necessary for successful exploitation. Early examples include CCFIR [75], BinCFI [77], kBouncer [54] and ROPecker [24]. Nonetheless, specialized CRAs were developed [22], [34], [35] breaking the aforementioned schemes, primarily due to their inabillity to fully eliminate ROP. A key distinction of PLATYPUS compared to prior schemes, particularly CCFIR [75], is its ability to fully restrict cross-DSO transfers at module granularity. For instance, CCFIR's Springboard section is global to the process; while it may limit cross-DSO function reachability, it does not do so per module, permitting a substantially broader range of targets for hijacked pointers. Moreover, CCFIR incurs significant performance overhead (3.6%-8.6%) in comparison to PLATYPUS.

In addition, modern coarse grained schemes currently deployed, like Intel's CET [40], Microsoft's Control Flow Guard [49], and ARM's BTI [11], although stronger, still fall short against sophisticated attacks like FOP [36], [44], [47]. With our defense in place, such attacks are mitigated and CET is significantly enhanced, as PLATYPUS provides the essential property needed to realistically block the key capability of CRAs, arbitrary cross-module transitions.

6.1.2. Fine-grained CFI. Fine-grained CFI schemes seek to raise the bar for CRAs by building restrictive CFGs, closely approximating program's actual control flow, making such attacks much harder. However, constructing precise CFGs in complex software is challenging, and their enforcement can incur significant runtime overhead (e.g., [30], [55]). Moreover, CFG errors must be avoided, as they can cause crashes even in benign executions, sometimes necessitating conservative decisions in the construction of target sets. Even with minimal target sets, security guarantees remain unclear and hard to evaluate. This is underscored by recent attacks capable of bypassing even fine-grained CFI schemes, such as COOP [64], CFB [23], and Control Jujutsu [31]. Furthermore, attacks like CFOP [12] demonstrate that CFI remains incomplete, highlighting CRAs as persistent threats.

Nonetheless, certain fine-grained schemes (e.g., [26], [13], [39]) significantly hinder exploitation while maintaining low overhead, making them attractive for adoption. However, on Intel architectures only LLVM Clang CFI [26] has seen real-world deployment, and even this remains limited [15]. A principal barrier is the inadequate modular support [10], [46]; many fine-grained proposals [26], [13], [52], [39], [42] cannot interoperate with unprotected DSOs, an inevitable scenario when integrating third-party or closed-source libraries[46]. Since constructed CFGs lack awareness of uninstrumented libraries, indirect calls may cause crashes or remain unprotected, hindering modular deployment. For instance, Clang CFI [26] avoids crashes but omits checks on calls to unprotected libraries, weakening CFI, especially if these libraries are vulnerable. Notably, glibc, which exposes numerous gadgets and critical APIs, remains unsupported by Clang CFI. This is a critical shortcoming, as most exploit chains invoke syscalls in glibc; leaving these transitions unprotected can undermine defenses elsewhere. In contrast, PLATYPUS securely coexists with unprotected modules while also effectively safeguarding glibc.

Binary-level CFI schemes [55], [75], [77], [71] offer broader applicability but weaker security and CFG precision than source-level counterparts [10]. Lockdown [55] is close to our work as it also restricts arbitrary indirect cross-DSO transitions. However, it relies on dynamic binary translation to enforce its CFG on runtime, incurring substantial overhead (19.09%). Moreover, it does not differentiate between imported symbols that are indirectly callable and those that

are not, leaving significant opportunities for attackers as shown in Table 5. In contrast, PLATYPUS closes this gap by introducing fake PLTs, with negligible overhead and no reliance on third-party components (e.g., libdetox).

6.2. Debloating

Software debloating enhances security and performance by removing unused code ("bloat") that expands the attack surface for CRAs. By reducing available gadgets, it impedes attacks. Debloating often relies on statically constructed CFGs to identify and eliminate unreachable code, producing a thinned program. It can also complement CFI by excluding code preserved by conservative CFGs, further limiting attacker capabilities. However, as debloating alone offers limited protection, it is most effective when combined with CFI. Many debloaters tailor programs using test cases [58], [38], while others remove undesired features based on configuration options [66], [9]. Debloating targets are also diverse, including Java applications and bytecode [20], [41], firmware [25], [76], and containers [61].

Debloating shared libraries. We focus on debloaters targeting shared libraries for C/C++ applications, as shared library functions are central to most CRAs and DSOs are typically rich in gadgets. Approaches focused on firmware [76] or server-side JavaScript [69] fall outside our scope.

Our analysis identifies a significant limitation of library debloaters: they compromise the fundamental property of system-wide sharing. Specifically, Quach et al. [59] specialize libraries per binary, creating new "library fragments" for every application and thereby undermining sharing. Nibbler [8] debloats libraries for a set of binaries, but the process must be repeated as new binaries are introduced, rendering it unsuitable for general-purpose systems. PieceWise [60] compiles libraries once but removes code at load time, potentially triggering *Copy on Write* (CoW); unless affected pages are fully eliminated, they are no longer shared. BlankIt [57] faces identical problems: its runtime instrumentation copies code from the original library to activate or deactivate functions on demand, again triggering CoW on shared pages.

This limitation renders library debloating schemes impractical for both memory-constrained and general-purpose systems, where shared libraries are prevalent. Hence, their adoption is hindered despite effectively eliminating gadgets. Additionally, PieceWise [60] suffers from "irreconcilable technical issues" during build and BlankIt [57] requires "time-consuming manual preprocessing" for new benchmarks [19]. In contrast, PLATYPUS does not remove code, but effectively "debloats" the scope of indirect transitions by restricting them from targeting external modules. This is achieved without requiring library recompilation or compromising their shared nature, thus preserving practicality.

6.3. Compartmentalization

Rather than emphasizing control-flow integrity, other approaches isolate components within a process or system to enforce privilege separation, typically relying on dedicated APIs for inter-component communication. ERIM [70] and Hodor [37] leverage Intel's PKU to separate trusted and untrusted components within a process, aiming to prevent sensitive data leakage. However, prior work [27] identified several PKU pitfalls, including inconsistent enforcement of PKU and page table permissions, both enabling bypasses of the aforementioned schemes. Additionally, PKU cannot provide code pointer integrity without incurring significant overhead [37]. Cali [14] isolates third-party libraries by placing each in a separate process; while effective, it is unsuitable for wide deployment within a system, since additional processes increase memory usage. Furthermore, it does not isolate standard libraries such as libc, thereby exposing critical functions in the presence of vulnerabilities. FlexOS [45], an operating system solution, provides intra-process isolation but requires developer annotations for complex indirect control-flow transitions, limiting its practicality. PLATYPUS, although not a compartmentalization scheme, prevents indirect transitions across module boundaries with minimal overhead, supports complex use cases such as callbacks, and protects crucial libraries without compromising security, all achieved transparently to the user.

Software Fault Isolation. SFI constitutes a specific form of compartmentalization in which safety boundaries are enforced in software by instrumenting the code of untrusted components. Prominent examples include WebAssembly [73], Native Client (NaCl) [72], and RLBox [51]. For instance, the primary goal of NaCl is to enable the safe execution of untrusted native C/C++ code within browser applications by confining it to sandboxed memory regions. In this way, potential vulnerabilities in untrusted code are contained within these restricted regions. Similarly, RLBox isolates third-party libraries in Mozilla Firefox using a comparable sandboxing approach. Both approaches enforce restrictions not only on the control flow of untrusted components but also on the set of permissible memory accesses. A key challenge lies in the secure and efficient enforcement of transitions between sandboxed and non-sandboxed components. Improper handling of such transitions can introduce vulnerabilities (as demonstrated in NaCl [5], [6]), while frequent transitions may incur significant performance overhead in applications with numerous such boundary crossings [51], [43].

Prior SFI schemes [72], [48], [56], [67] have employed similar masking techniques to enforce the boundaries of untrusted components. Unlike classic SFI, the goal of PLATYPUS is not general-purpose memory sandboxing, but rather the restriction of indirect control-flow transfers to intra-DSO targets. As discussed in § 2.6, the novelty of our approach lies in combining this masking technique with PLTs to strengthen forward-edge protection in schemes such as CET,

thereby mitigating the longstanding issue of unrestricted cross-module transitions.

7. Conclusion

We propose PLATYPUS, a practical defense that significantly restricts cross-DSO indirect transitions in CET-enabled systems. Our analysis and experiments demonstrate substantial security gains for CET-instrumented binaries with minimal performance overhead. Furthermore, PLATYPUS remains compatible with unprotected and dynamically loaded libraries, making it suitable for deployment in real-world environments.

8. Ethics considerations

None.

9. LLM usage considerations

LLMs were used for editorial purposes in this manuscript, and all outputs were inspected by the authors to ensure accuracy and originality.

References

[1] Libselinux source code, selinux_restorecon.c. https://github.com/SELinuxProject/selinux/blob/main/libselinux/src/selinux_restorecon.c#L1302. (14-04-2026).

[2] Memcached source code, assoc.c. https://github.com/memcached/memcached/blob/master/assoc.c#L278. (14-04-2026).

[3] Memcached source code, proto_text.c. https://github.com/memcached/memcached/blob/master/proto_text.c#L791. (14-04-2026).

[4] Nginx source code, ngx_thread_pool.c. https://github.com/nginx/nginx/blob/master/src/core/ngx_thread_pool.c#L157. (14-04-2026).

[5] Security: NaClSwitch() leaks NaClThreadContext pointer to x86-32 untrusted code. https://issuetracker.google.com/issues/42402545. (31-03-2026).

[6] Uninitialized sendmsg syscall arguments in sel_ldr. https://issuetracker.google.com/issues/42400629. (31-03-2026).

[7] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, 2005.

[8] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. Nibbler: debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019.

[9] Mohannad Alhanahnah, Rithik Jain, Vaibhav Rastogi, Somesh Jha, and Thomas Reps. Lightweight, Multi-Stage, Compiler-Assisted Application Specialization . In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pages 251–269. IEEE Computer Society, 2022.

[10] Mahmoud Ammar, Adam Caulfield, and Ivan De Oliveira Nunes. SoK: Integrity, Attestation, and Auditing of Program Execution . In *2025 IEEE Symposium on Security and Privacy (SP)*, 2025.

[11] ARM. *ARM A64 Instruction Set Architecture - Branch Target Identification*, 2020.

[12] Marcos Sanchez Bajo and Christian Rossow. Await() a Second: Evading Control Flow Integrity by Hijacking C++ Coroutines. In *USENIX Security*, Seattle, WA, 2025.

[13] Markus Bauer, Ilya Grishchenko, and Christian Rossow. TyPro: Forward CFI for C-Style Indirect Function Calls Using Type Propagation. In *Proceedings of the 38th Annual Computer Security Applications Conference*, 2022.

[14] Markus Bauer and Christian Rossow. Cali: Compiler-Assisted Library Isolation. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021.

[15] Lucas Becker, Matthias Hollick, and Jiska Classen. SoK: on the effectiveness of control-flow integrity in practice. In *Proceedings of the 18th USENIX Conference on Offensive Technologies*, 2024.

[16] Lorenzo Binosi, Gregorio Barzasi, Michele Carminati, Stefano Zanero, and Mario Polino. The Illusion of Randomness: An Empirical Analysis of Address Space Layout Randomization Implementations. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024.

[17] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, 2011.

[18] Erik Bosman and Herbert Bos. Framing Signals - A Return to Portable Shellcode. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 2014.

[19] Michael D. Brown, Adam Meily, Brian Fairservice, Akshay Sood, Jonathan Dorn, Trail of Bits, and Ronald Eytchison. A broad comparative evaluation of software debloating tools. In *Proceedings of the 33rd USENIX Conference on Security Symposium*, 2024.

[20] Bobby R. Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. JShrink: in-depth investigation into debloating modern Java applications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, 2020.

[21] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 2018.

[22] Nicholas Carlini and David Wagner. ROP is still dangerous: breaking modern defenses. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, 2014.

[23] Nicolas 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 Conference on Security Symposium*, 2015.

[24] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H. Deng. ROPecker: A Generic and Practical Approach For Defending Against ROP Attacks. In *NDSS*, 2014.

[25] Jake Christensen, Ionut Mugurel Anghel, Rob Taglang, Mihai Chiroiu, and Radu Sion. DECAF: automatic, adaptive de-bloating and hardening of COTS firmware. In *Proceedings of the 29th USENIX Conference on Security Symposium*, SEC'20, USA, 2020. USENIX Association.

[26] Clang/LLVM. Control Flow Integrity Design Documentation. https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html. (12-11-2025).

[27] 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 Conference on Security Symposium*, 2020.

[28] Glibc developers. IO_validate_vtable (Glibc 2.41 Source Code). https://elixir.bootlin.com/glibc/glibc-2.41.9000/source/libio/libioP.h#L1033. (12-11-2025).

[29] Nginx developers. Test suite for nginx. https://github.com/nginx/nginx-tests. (30-03-2026).

[30] Ren Ding, Chenxiong Qian, Chengyu Song, William Harris, Taesoo Kim, and Wenke Lee. Efficient protection of path-sensitive control security. In *Proceedings of the 26th USENIX Conference on Security Symposium*, 2017.

[31] 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*, 2015.

[32] Alexander J. Gaidis, Joao Moreira, Ke Sun, Alyssa Milburn, Vaggelis Atlidakis, and Vasileios P. Kemerlis. FineIBT: Fine-grain Control-flow Enforcement with Indirect Branch Tracking. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, 2023.

[33] Will Glozer. wrk: Modern HTTP benchmarking tool. https://github.com/wg/wrk. (12-11-2025).

[34] 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 Conference on Security Symposium*, 2014.

[35] 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*, 2014.

[36] Yingjie Guo, Liwei Chen, and Gang Shi. Function-Oriented Programming: A New Class of Code Reuse Attack in C Applications. In *2018 IEEE Conference on Communications and Network Security (CNS)*, 2018.

[37] 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 Conference on Usenix Annual Technical Conference*, 2019.

[38] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018.

[39] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing unique code target property for control-flow integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, New York, NY, USA, 2018. Association for Computing Machinery.

[40] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual.*, 2020.

[41] Christian Gram Kalhauge and Jens Palsberg. Binary reduction of dependency graphs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[42] Mustakimur Rahman Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. Origin-sensitive control flow integrity. In *Proceedings of the 28th USENIX Conference on Security Symposium*, 2019.

[43] Matthew Kolosick, Shravan Narayan, Evan Johnson, Conrad Watt, Michael LeMay, Deepak Garg, Ranjit Jhala, and Deian Stefan. Isolation without taxation: near-zero-cost transitions for webassembly and sfi. *Proc. ACM Program. Lang.*, 6(POPL), January 2022.

[44] Bingchen Lan, Yan Li, Hao Sun, Chao Su, Yao Liu, and Qingkai Zeng. Loop-Oriented Programming: A New Code Reuse Attack to Bypass Modern Defenses. In *2015 IEEE Trustcom/BigDataSE/ISPA*, 2015.

[45] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. FlexOS: towards flexible OS isolation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022.

[46] Yuan Li, Mingzhe Wang, Chao Zhang, Xingman Chen, Songtao Yang, and Ying Liu. Finding Cracks in Shields: On the Security of Control Flow Integrity Mechanisms. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[47] LMS. Bypassing CET & BTI With Functional Oriented Programming. https://phrack.org/issues/71/7_md#article, (19-08-2024).

[48] Stephen McCamant and Greg Morrisett. Evaluating sfi for a cisc architecture. In *USENIX Security Symposium*, 2006.

[49] Microsoft. Control Flow Guard for platform security. https://learn.microsoft.com/en-us/windows/win32/secbp/control-flow-guard. (12-11-2025).

[50] musl libc team. musl libc: Open Issues. https://wiki.musl-libc.org/open-issues. (12-11-2025).

[51] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting fine grain isolation in the firefox renderer. In *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[52] Ben Niu and Gang Tan. Modular control-flow integrity. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[53] Carlos O'Donell. The GNU C Library version 2.34 is now available. https://sourceware.org/pipermail/libc-alpha/2021-August/129718.html. (12-11-2025).

[54] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In *Proceedings of the 22nd USENIX Conference on Security*, 2013.

[55] Mathias Payer, Antonio Barresi, and Thomas R. Gross. Fine-Grained Control-Flow Integrity Through Binary Hardening. In *Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9148*, 2015.

[56] Gregor Peach, Runyu Pan, Zhuoyi Wu, Gabriel Parmer, Christopher Haster, and Ludmila Cherkasova. ewasm: Practical software fault isolation for reliable embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[57] Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. BlankIt library debloating: getting what you want instead of cutting what you don't. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.

[58] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. RAZOR: a framework for post-deployment software debloating. In *Proceedings of the 28th USENIX Conference on Security Symposium*, 2019.

[59] Anh Quach and Aravind Prakash. Bloat factors and binary specialization. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*, FEAST'19, New York, NY, USA, 2019. Association for Computing Machinery.

[60] Anh Quach, Aravind Prakash, and Lok Yan. Debloating software through piece-wise compilation and loading. In *Proceedings of the 27th USENIX Conference on Security Symposium*, 2018.

[61] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. Cimplifier: automatically debloating containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[62] RedisLabs. memtier_benchmark: NoSQL Redis and Memcache traffic generation and benchmarking tool. https://github.com/RedisLabs/memtier_benchmark. (12-11-2025).

[63] Giampaolo Rodola. pyftpdlib: Extremely fast and scalable Python FTP server library. https://github.com/giampaolo/pyftpdlib/blob/master/scripts/ftpbench. (12-11-2025).

[64] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, 2015.

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

[66] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. Trimmer: application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[67] Christopher Small. A tool for constructing safe extensible c++ systems. In *Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS 97)*, 1997.

[68] Ubuntu Security Team. Security Features in Ubuntu. https://wiki.ubuntu.com/Security/Features. (12-11-2025).

[69] Alexi Turcotte, Ellen Arteca, Ashish Mishra, Saba Alimadadi, and Frank Tip. Stubbifier: debloating dynamic server-side JavaScript applications. *Empirical Softw. Engg.*, 2022.

[70] 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 USENIX Conference on Security Symposium*, 2019.

[71] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *2016 IEEE Symposium on Security and Privacy (SP)*, 2016.

[72] Bennet Yee, David Sehr, Greg Dardyk, Brad Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy (Oakland'09)*, 2009.

[73] Alon Zakai, Andreas Haas, Andreas Rossberg, Ben Titzer, Dan Gohman, Derek Schuff, JF Bastien, Luke Wagner, and Michael Holman. Bringing the web up to speed with webassembly. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[74] Adhemerval Zanella. Glibc. https://github.com/zatrazz/glibc/tree/azanella/clang. (04-09-2024).

[75] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical Control Flow Integrity and Randomization for Binary Executables. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, 2013.

[76] 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 ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, New York, NY, USA, 2022. Association for Computing Machinery.

[77] Mingwei Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Conference on Security*, 2013.

Appendix A. PLATYPUS Corner Cases

Apart from the corner cases presented in § 3.4 and § 3.5, there are a few more scenarios that do not fit into the standard callback or struct scenarios previously discussed.

Table 5. REDUCTION OF INDIRECTLY ACCESSIBLE PLTS. COL.2: # PLTS IN THE UNMITIGATED MODULE. COL. 3: # FAKE PLTS UNDER PLATYPUS PROTECTION.

	Total PLTs	Fake PLTs	Red. (%)
redis-server	248	6	97.58
nginx	368	6	98.37
sqlite3	115	43	62.61
libc.so	13	0	100.00
libpcre2-8.so	13	0	100.00
libz.so	17	0	100.00
libcrypto.so	152	1	99.34
libssl.so	549	17	96.90
libtinfo.so	64	3	95.31
libreadline.so	104	1	99.03
libncurses.so	107	3	97.19
libcrypt.so	23	21	8.70
libm.so	8	0	100.00

A.1. Initialization and Finalization Functions

Many ELF binaries contain dedicated sections, such as *.init* and *.fini*. These sections encapsulate instructions to be executed, respectively, prior and subsequent to the invocation of the main function. Their purpose is to guarantee correct initialization and termination procedures for the binary, including the management of global constructors and destructors. Traditionally, these two sections have been relied upon; however, on modern ELF platforms, the more versatile *.init_array* and *.fini_array* sections are preferred. The principal distinction between them is that the latter sections store arrays of function pointers, each to be executed during program initialization or finalization, respectively.

Special consideration must be given to these four sections, as their functions or referenced pointers are indirectly invoked by glibc (specifically via the *call_init* and *_dl_call_fini* functions) during program startup and termination. Consequently, any masking applied at the relevant invocation points within glibc transforms these pointers into intra-module addresses, resulting in corruption, as they reference locations within other modules However, in both cases, the precise set of function pointers that will be invoked is statically known, since this information is already encoded within the aforementioned sections of each ELF's metadata. Therefore, these corner cases can be addressed by creating a dedicated callback table for these functions, thereby eliminating indirect calls and ensuring that only the intended pointers may be invoked. We have already discussed this regarding *_dl_call_fini* in § 4.2.2.

A.2. Main and Exit Handlers

The same approach described in § A.1 is also applicable to the cases involving the *__libc_start_call_main* and *__run_exit_handlers* functions in libc. The former indirectly invokes the *main* function of the program, while the latter is responsible for indirectly invoking functions registered via

Table 6. BENCHMARK DEPENDENCIES: COL.2 SHOWS THE NUMBER OF DSO DEPENDENCIES, AND COL.3 THE NUMBER LOADED VIA `DLOPEN`.

Benchmark	# DSO Deps.	# DSOs loaded via `dlopen`
rm	1	0
uniq	1	0
date	1	0
mkdir	3	0
sort	2	0
chown	6	5
gzip	1	0
grep	2	0
tar	7	4
bzip2	1	0
bftpd	6	4
make	3	2
lighttpd	4	0
objdump	3	0
memcached	2	0
wget	10	2
Nginx	6	0
Redis	5	1
SQLite	6	0

Table 7. INCREASE IN BINARY SIZE OF INSTRUMENTED LIBRARIES

	Instrumented (KB)	Original (KB)	Increase (%)
libc.so	1901	1843	3.15
libpcre2-8.so	387	383	1.04
libz.so	106	102	3.92
libevent-2.1.so	379	341	11.14
libcrypto.so	5260	5137	2.39
libssl.so	978	933	4.82
libunistring.so	1934	1897	1.95
libuuid.so	29	28	3.57
libidn2.so	193	191	1.05
libm.so	917	935	-1.93
libselinux.so	205	189	8.47
libtinfo.so	252	184	36.95
libreadline.so	393	367	7.08
libncurses.so	218	204	6.86
libcrypt.so	215	212	1.42
libzstd.so	837	832	0.60

atexit calls. Our symbol gathering pass (see § 3.4.2) already identifies these registered functions. Although it would be possible to dedicate specific callback tables to each of these two functions, we opted to include the corresponding symbols in libc's callback table and to instrument the associated functions such that, instead of making indirect calls, they directly invoke the PLT corresponding to the relevant special function in the main binary (see § 3.4.3). This approach was chosen primarily because, in typical scenarios, the number of added symbols is minimal (e.g., there is only a single *main*). However, PLATYPUS can also support dedicated callback tables for these functions; the choice is left to the user. Importantly, these interventions—including those described in § A.1—do not compromise the generality of our approach, as they are necessary for the correct operation of all programs (dependent on glibc).

A.3. Stack Unwinding

Backtrace and stack unwinding functionalities are primarily significant for debugging purposes, such as crash diagnostics, and for exception handling. Given that these features may be required in certain programs, we extend PLATYPUS to support them. For instance, within the Redis testsuite, certain tests involve sanitizers producing stack traces. In glibc, the *__backtrace* and *backtrace_helper* functions are invoked for this purpose, which subsequently make indirect calls to routines located in libgcc (e.g. *_Unwind_Backtrace*). To accommodate this workflow, we instrument the former functions with the extended masking mechanism and additionally include the latter in libc's callback table. Notably, this extension to the callback table is necessary only for programs that explicitly require stack backtracking functionality.

A.4. Callbacks from Unprotected Modules

When handling uninstrumented libraries, a particular case that requires special attention arises when such libraries use external CBGs (see § 3.4), passing as CPTRs pointers to themselves. Since PLATYPUS has not analyzed the unprotected module with our symbol gathering pass, we are unable to add the corresponding CPTR to the callback table of the DSO containing the CBG. This issue can be alleviated through binary analysis, as in many cases CPTRs are passed directly as symbols rather than as variables in the arguments of the CBG. To fully eliminate the possibility of a crash, we can modify the callback table of *every* module such that if the pointer in `r11` references an unprotected module, the indirect transition is always allowed. This modification would permit only the indirect transitions instrumented with the extended masking mechanism to reach arbitrary functions within unprotected modules. However, given our CPTRs check on the caller-side (see "*Arguments to CBGs.*" in § 5), it would still remain impossible for a protected module to invoke a CBG with specifically chosen arguments (e.g. a vulnerable unprotected function). We preferred this design over allowlisting unprotected modules in every indirect transition as happening currently in Clang CFI [26].

Table 8. INCREASE IN BINARY SIZE OF INSTRUMENTED BENCHMARKS

	Instrumented (KB)	Original (KB)	Increase (%)
date	97	88	10.22
chown	76	65	16.92
uniq	51	43	18.60
mkdir	83	69	20.29
rm	74	64	15.63
sort	134	116	15.52
grep	190	155	22.58
bzip2	119	113	5.31
tar	480	430	11.63
gzip	108	98	10.20
make	259	250	3.60
objdump	2920	2856	2.24
memcached	279	252	10.71
bftpd	98	83	18.07
lighttpd	610	595	2.52
wget	497	456	8.99
nginx	1255	1214	3.38
redis-server	3559	3497	1.77
sqlite3	1850	1750	5.71

Appendix B.
Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

The paper presents a new Control Flow Integrity system called PLATYPUS with a granularity between coarse-grained and fine-grained CFI. Coarse-grained CFI systems such as Intel CET allow an indirect call to target any function in a program, leaving a significant percentage of the program's attack surface unprotected. Platypus improves the security of Intel CET-based CFI systems by restricting an indirect call strictly to targets within the same module, thus improving the security.

B.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Addresses a Long-Known Issue
- Provides a Valuable Step Forward in an Established Field

B.3. Reasons for Acceptance

1) The paper creates a new tool to enable future science. PLATYPUS is a new tool that advances CFI enforcement. PLATYPUS restricts indirect calls to intra-module targets using lightweight pointer masking on top of Intel CET. The paper handles corner-cases such as callbacks and dynamically loaded libraries, making it practical. The prototype is also available open-source.
2) The paper addresses a long-known issue. All reviewers agree that hardening cross-DSO indirect calls is an important problem. PLATYPUS's module-level restriction presents a practical solution, with low performance overhead. PLATYPUS is shown to reduce indirectly accessible cross-DSO functions by over 98% with a minimal overhead of 0.5%.

B.4. Noteworthy Concerns

The code coverage achieved by PLATYPUS is not reported.