Renewable Just-In-Time Control-Flow Integrity

Erick Bauman
The University of Texas at Dallas, USA

Jun Duan
The University of Texas at Dallas, USA

Kevin W. Hamlen
The University of Texas at Dallas, USA

Zhiqiang Lin
The Ohio State University, USA

ABSTRACT

RENEW (REwriting Newly Executable pages after Writes) unites and extends recent advances in binary code analysis and transformation to solve a longstanding compatibility problem for binary code security hardening algorithms—support for arbitrary dynamically self-modifying code. Self-modification is now a mainstay of many consumer software products, including Just-In-Time (JIT) compiled languages, on-demand component loading, self-extracting installers, and self-hooking APIs; but it poses significant challenges for code hardening algorithms that rely on computationally heavy static analyses, source code information, or compiler-specific code generation patterns. As a result, many of the strongest protection mechanisms for code hardening have remained incompatible or significantly weakened for the large class of software that incorporates self-modification (either directly or within its underlying runtime systems).

By leveraging recent advances in lightweight binary disassembly, efficient memory page interception, and fast machine code rewriting, RENEW transparently extends binary code security hardening algorithms, such as source-free control-flow integrity (CFI) and software fault isolation (SFI), to self-modifying target codes. Experiments on two commodity JIT compilers and a commodity self-extracting installer solution show that RENEW supports highly diverse dynamic code generation strategies with little or no customization to each new application, and achieves a 3–4× performance improvement over alternative solutions that disable dynamic code to achieve equivalent security guarantees.

CCS CONCEPTS

• **Security and privacy → Web application security**; **Software security engineering**.

KEYWORDS

runtime code generation (RTCG), virtual machines (VMs)

ACM Reference Format:
Erick Bauman, Jun Duan, Kevin W. Hamlen, and Zhiqiang Lin. 2023. Renewable Just-In-Time Control-Flow Integrity. In *The 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '23), October 16–18, 2023, Hong Kong, China.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3607199.3607239

RAID '23, October 16–18, 2023, Hong Kong, China

ACM ISBN 979-8-4007-0765-0/23/10.
https://doi.org/10.1145/3607199.3607239

1 INTRODUCTION

Dynamically generated code is an increasingly popular strategy for solving performance and compatibility problems in large, commercial software products. Nearly all modern web browsers rely upon dynamically JIT-compiled code to achieve competitive performance for web scripts (e.g., JavaScript). Similarly, all software that runs atop the Microsoft .NET architecture, and most software written in Java or that incorporates the Component Object Model (COM) standard, relies on dynamically generated code within its runtime system. More than 70% of the top programming languages are JIT-compiled [64], using myriad different code generation algorithms and rapidly evolving optimization strategies.

JIT compilers are just one form of this increasingly ubiquitous computing paradigm. For example, self-extracting archive technologies (e.g., UPX [51]) reduce space overheads of large, multi-component applications by compressing the code of lesser used modules and unpacking them into the memory address space on demand. On-demand loading and linking of self-extracting software components has become a standard feature of Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS) cloud architectures. Aggressively optimized software within these ecosystems employs non-standard self-hooking strategies to dynamically detect and modify its own code to efficiently incorporate new components [73].

Unfortunately, the advantages of dynamically generated code come at a steep price for security: the fluidity and diversity of dynamic code make it difficult to defend, making it a prime target for adversaries. For example, over half of "in the wild" Chrome exploits from 2018–2021 abused JIT bugs [53], and at least 37% of malware threats conceal themselves during propagation through self-unpacking [63]. The risks exposed by dynamically generated code also introduce entirely new classes of attacks, such as JIT spraying [7] and deep packing [62].

One reason why dynamic code is so difficult to defend is that it remains incompatible with many of the most powerful and efficient code protection approaches. For example, source-level vulnerability detectors (e.g., [18, 46, 55]) and binary static analyses (e.g., [9, 11, 34, 66]) generally cannot examine or diagnose dynamically generated binary code. Binary code hardening approaches, such as *control-flow integrity* (CFI) [1] and *software fault isolation* (SFI) [65], offer greater power [26], but most CFI and SFI implementations must assume target codes have strictly non-executable data (NXD), making them incompatible with self-modifying applications.

In the specific case of web browsers, prior work has shown that individual JIT compilers can be manually modified to generate CFI-compliant code, yielding an ideal balance of high performance and strong security [48]. However, this strategy has proved difficult to maintain in practice as JIT compilers have increased in complexity and diversity. For example, many of the top JIT engines, including

Chrome V8, still use RWX memory pages to achieve high performance [49], which precludes efficient CFI technologies like Intel CET [3]. As a result, these solutions have remained incompatible and unimplemented for any of the top browsers of the past 8 years; vendors currently recommend disabling the JIT engine entirely to enable CFI security [50], losing all performance and compatibility benefits of dynamic code.

RENEW explores an alternative solution that implements a *dynamically renewable* form of CFI for arbitrary self-modifying codes. Our goal is to provide developers a means of securing dynamically self-modifying products with strong binary protections like CFI without abandoning dynamic code or limiting their consumers to a custom architecture (e.g., VM or OS). We find that recent advances in efficient, security-resilient disassembly [5] and robust binary reassembly [68] now make it feasible to rapidly disassemble, instrument, and reassemble dynamic code pages with CFI protections at runtime in a way that is agnostic to the target application's specific strategy for generating and modifying its code.

While less efficient than a manually CFI-instrumented JIT compiler or unpacker, our solution has the benefit of requiring little or no change to support new forms of dynamic code (e.g., new JIT compiler versions), and occupies a sweet spot between disabling dynamic code entirely and leaving it unsecured. To reach this sweet spot, we propose a layered defense that achieves highest performance when lightweight static-dynamic code analysis successfully predicts control-flow transfers requiring guards, and falls back to a slower but higher assurance mediation when necessary. The result is a balance of high performance without compromising security.

In summary, we make the following contributions:

- RENEW is the first CFI framework designed to support arbitrary dynamically generated code.
- We provide a robust, source-agnostic method of intercepting and rewriting dynamic code that combines recent innovations in binary disassembly and reassembly.
- Through experiments on over 6000 JIT-compiled and self-unpacking binaries across three programming languages (JS, Lua, and C), we develop a memory page management algorithm that preserves non-malicious application functionalities while securing dynamically generated code.
- The flexibility and applicability of RENEW is evaluated on two large JIT compilers (LuaJIT and Spidermonkey) and an executable packer (UPX).
- We find that JIT+RENEW is faster than disabling the JIT to enable CFI, with LuaJIT and Spidermonkey being 3.4× and 3.9× faster, respectively.

2 BACKGROUND AND OVERVIEW

Prior efforts to harden dynamic code have mainly focused on code produced by specific JIT compiler versions, and take the approach of customizing the code producer rather than modifying binary code after it is produced [4, 21, 48, 60, 75]. This typically requires a deep understanding of the JIT compiler, and can therefore be brittle across compiler version changes. Since anticipating all dynamic code produced by arbitrary target programs is undecidable in the general case [13], any more general solution faces the daunting challenge of incorporating a full binary disassembly, rewriting, and reassembly framework into the target application, so that it can self-secure arbitrary dynamically generated code on the fly.

Until recently, the overheads associated with such renewable rewriting have been prohibitive because most binary disassembly, rewriting, and reassembly solutions rely upon computationally heavy code analyses (e.g., code reachability analysis, symbolic execution, abstract interpretation, pointer-integer disambiguation, code-data disambiguation, etc.) that are designed to be conducted statically, and that therefore sacrifice time and space efficiency to achieve high correctness and completeness for generated code. Dynamic binary rewriters (e.g., [6, 8, 45]) bypass control-flow decidability problems by interpreting, shepherding, or mediating untrusted code at instruction or block granularity, but can incur overheads of hundreds or even thousands of percents. Static rewriting is typically faster, exhibiting overheads of 10% or less [10], but lacks precision since it cannot decide recursively enumerable properties of the control-flow graph (CFG) [26, 36].

However, the advent of lightweight yet powerful binary analyses such as *superset disassembly* [5] and *reassembleable disassembling* [68] have opened new opportunities for implementing fast static rewriting within a dynamic framework. Superset disassemblers avoid the computationally heavy (and generally undecidable) task of recovering a single correct disassembly from an input byte sequence by instead recovering a superset of all the possible disassemblies without deciding which is the correct one. All but one of the disassemblies in the recovered superset might be unreachable (i.e., dead code), but code hardening defenses can conservatively harden all paths in the superset to guaranteeably protect all reachable paths. Reassembleable disassemblies allow the modified, disassembled code to be quickly reassembled back into executable machine instructions without requiring a full compiler.

RENEW leverages these new capabilities to introduce a new hybrid static-dynamic approach that is *generational*, dynamically applying multiple rounds of static rewriting. A static binary rewriter in-lined into the target program intercepts and rewrites any dynamically generated code, with no further intervention unless the dynamic code is further modified or generates new dynamic code. Dynamic modifications solicit additional rounds of rewriting, freeing stale generations of rewritten pages to reduce space overhead. This cleanly separates the generated code into insecure freshly generated code that needs rewriting before it can be permitted to execute, and rewritten secure code that can be safely executed without (further) modification.

2.1 JIT Compilers

JIT compilers improve the performance of interpreted languages that are often too flexible to compile directly to native machine code. They bridge this gap by compiling fragments of the code as it runs. This allows the JIT compiler to repeatedly revise the generated assembly code as the interpretation progresses.

For example, polymorphic languages pass variables of various types to callees. The compiler might generate code for a specific native data type the first time the function is used, but implement a hook that calls back into the JIT compiler if the callee later receives a new type of argument. At that point it can revise the generated code to handle the new data type.

JIT code therefore undergoes continuous change, with bidirectional control-flow edges between code-generating and generated code, and multiple tiers of code generation. Lightweight generation is employed first, with more complex and expensive generation to optimize hot regions later. This complexity makes JIT compilers subject to rapid churn.

2.2 Executable Packers

Executable packers reduce the size of applications and code on disk and in memory; but unlike generic compression, they create compressed, self-unpacking, executable code with the same functionality as the original application. Securing arbitrary, untrusted binary code that dynamically unpacks itself on-demand is beyond the capabilities of existing CFI solutions.

UPX [51] is a prominent binary packer that supports many binary formats. When packing code for most ISAs (e.g., `linux/elf386`), it produces a binary that dynamically self-modifies, decompressing executable code directly into memory without writing the expanded version to disk. Packed binaries contain unpacking code that decompresses the original binary into memory before transferring control to the decompressed image.

The UPX source code utilizes a stub written in a combination of C and assembly code to decompress the binary payload. Most of the decompression stub is also compressed, leaving only a tiny amount of initialization and decompression code uncompressed. The compressed binary therefore unpacks in two stages, first unpacking and jumping to the full decompressor before unpacking the actual binary and setting up its environment. This style of dynamic code generation is therefore substantially different than that implemented by JIT compilers, with fewer generations of dynamically generated code but no consistency or recognizable patterns in the generated code (which can be completely arbitrary).

2.3 Challenges

2.3.1 Intercepting Dynamically Generated Code. Handling arbitrary dynamically generated code poses several research challenges. Solutions that anticipate a finite number of code-generation phases are inadequate, since self-modification can occur continuously throughout the program's lifetime. In addition, self-modifying code may change its own bytes as it executes, blurring the line between generating and executing the code. This is a worst-case scenario for protecting dynamic code, as every memory write could potentially invalidate the next instruction to be executed.

To strike a useful balance between performance and compatibility, we choose to target applications whose memory writes are not to the same virtual memory page as the current program counter. Same-page code-rewriting is slow and therefore rare outside of malware [57], making this choice compatible with almost all non-malicious dynamic code, while halting (or crashing) most self-obfuscating malware. It also allows us to partition self-modifying program traces into alternating *generating* and *executing* phases.

A related challenge involves memory regions that are both writable and executable. Despite being strongly discouraged by security professionals, such regions are still commonplace in commercial software. RENEW protects such regions by deferring the target application's requests for writable and executable page access permissions. During generating phases, writes revoke execute permissions, which are deferred to the next executing phase; and during executing phases, executions revoke write permissions, which are deferred to the next generating phase.

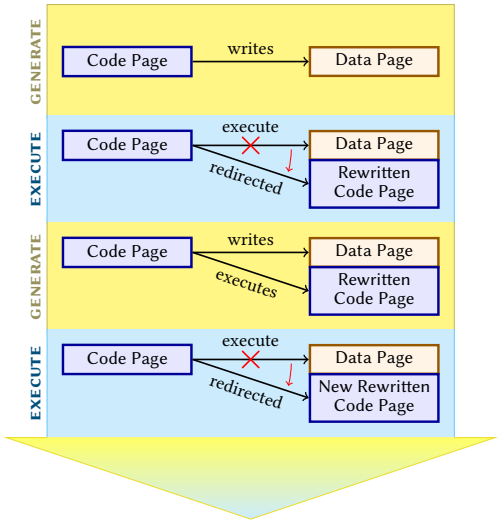

Figure 1: Dynamic code page life cycle

Figure 1 illustrates how this forces programs that perform unprincipled self-modification into distinct, alternating phases, and prevents any region from being simultaneously writable and executable. While in the executing phase, attempts to execute generated code are redirected to rewritten, hardened versions of each code region. If the program later attempts to write to read-only code, we transition back to a generating phase, set the page writable, and remove any now-obsolete hardened code (which will be re-rewritten next phase). This can continue for arbitrary code generations.

Interception and redirection of attempts to execute dynamic code has two implementations: a *fast path* and a *slow path*. When code hardening can predict all targets of a dynamically generated control-flow transfer instruction, it replaces the instruction with a fast-path instruction sequence that directly targets the executable, hardened code page (generating it on-demand if necessary). Otherwise, the transfer triggers RENEW's signal handler, which intercepts attempts to execute non-rewritten (and therefore non-executable) code pages for which the prediction failed. This is the slow path.

2.3.2 Rewriting Dynamically Generated Code. Guarding indirect control-flow transfer instructions, which compute their targets from runtime data, is the central challenge for enforcing CFI, SFI, and other sandboxing policies. While strictly static rewriters can perform complex binary analyses that differentiate code from data and in-line trampolines without corrupting or displacing data (e.g., [71, 72]), we found that such approaches are too slow or unreliable to accommodate unconstrained dynamic codes. For example, some JIT compilers read the code bytes they dynamically generate (e.g., operands of generated `mov` instructions), to guide subsequent data relocations. This means that overwriting any original bytes risks incompatibility, breaking most of the established approaches.

RENEW therefore tracks correspondences between original and rewritten code pages by storing code pointer tables at fixed offsets from all dynamically generated code addresses. This allows fast-path (and slow-path) redirection code to quickly locate and target appropriate hardened code pages at runtime without corrupting any of the originally generated code/data bytes, which remain available for reading. To accommodate densely packed jump targets and disassembly uncertainty, the lookup table contains a distinct pointer for every byte in the unsecured dynamically generated code.

Storing code lookup tables at fixed offsets could potentially pose problems for target applications that demand large memory regions at fixed virtual addresses, resulting in memory allocation conflicts. However, the prevalence of rebasable libraries and flexible memory management makes such requirements rare; all software we tested makes any fixed memory reservations early during its execution. By waiting until after the first region has been created, we can dynamically select a fixed offset that avoids conflicts.

Back-edges from dynamic code to non-dynamic code are facilitated by word-aligning all jump targets in rewritten code pages, which frees up the low-order bits of all pointers to dynamic code. The low-order bits are repurposed as a tag that alerts the CFI guard code to expedite a jump to a static code page rather than a rewritten page. This improves performance without relying upon any particular style of code generation in the target code.

2.3.3 Position-independent Code. Another challenge involves conflation of call-return control-flows with position-independent code (PIC) pointer computations. On Intel x86 architectures, PIC is typically implemented by executing a dummy call instruction whose purpose is to push the program counter onto the stack rather than calling a subroutine. Redirecting these special call instructions as if they are call-return flows breaks PIC, but in general they can be indistinguishable from call-returns. This is a difficult source of incompatibilities for many binary code rewriters.

We therefore take the approach of replacing all call instructions in dynamic code with push-jump sequences that push the stale return address of the original, unhardened code rather than the replacement, hardened code that is actually executing. This preserves PIC computations that expect to locate data at relative offsets to the original code. To preserve call-return flows, return instructions are replaced with the same instruction sequence that facilitates redirection of other forms of control-flow transfer. This introduces a small performance overhead (since hardware call-return prediction is less effective) to preserve compatibility.

2.4 Threat Model

RENEW is designed to augment arbitrary code hardening implementations (e.g., CFI) with support for arbitrary self-modifying code in target applications in a way that is agnostic to the target application's code generation strategy. It therefore hardens only the dynamically generated code, leaving the static code to be hardened in the typical way prior to execution (e.g., through application of static CFI), which can be performed with existing source-assisted or binary-only solutions. This static hardening must additionally insert the initialization code for RENEW that begins the dynamic interposition and instrumentation process for the dynamic code prior to any execution of dynamically generated code.

Our defense enforces *control-flow policies*, which are defined in the literature as (possibly dynamic) CFGs that whitelist the permissible edges traversed by executions of the target code. Control-flow policies are a foundation for enforcing other computable safety policies [26] (e.g., memory safety [2]) through in-lined reference monitors (IRMs) [56]. For example, RENEW prevents corruption of its data pages by setting them non-writable and enforcing a CFI policy that constrains all memory API function calls and system traps that could remove the write protections. The guard code that RENEW in-lines around all possible memory API calls constitutes an IRM whose complete mediation is assured by the CFI enforcement. Safety of this approach has been established in the prior literature by a dovetailed inductive argument [2]: Any successful attack must have a first policy violation, but corrupting a protected data page requires first hijacking the control-flow to circumvent the write protections, and hijacking the control-flow requires first corrupting the data page that constrains the flow. Since neither form of violation can be first, no first violation is possible.

In contrast to prior works, RENEW enforces these policies on arbitrary self-modifying codes that can consist of any (possibly even malicious) instruction sequences—even those that might not have been generated by a compiler. It does so by computing a conservative superset of all possible execution paths through the dynamic code, and brute-force guarding all possibly policy-violating instructions with IRM guard code. However, the assurance provided by RENEW is limited to control-flow policies as defined above. Higher-level safety policies, such as file system integrity or network protocol adherence, can potentially be enforced using RENEW as a foundation for guarding all policy-relevant operations (e.g., API calls) with uncircumventable IRMs, and is reserved for future work.

Our research does not concern the problem of formulating appropriate control-flow policies to enforce, or innovate a new static CFI. Rather, we take as given the policy and an appropriately robust and powerful static CFI algorithm for enforcing it, and our goal is to extend that same level of protection to the dynamic code.

While RENEW targets arbitrary generated code, our prototype only supports applications that do not override signal handlers. Future implementations can support signal-handling applications by intercepting signal handler registrations and re-inserting RENEW's handler before application-provided handlers in the handler chain.

To demonstrate generality, our prototype implementation targets Intel x86 binaries, which constitute an especially difficult yet ubiquitous ISA for CFI. Supporting 32-bit processes on 64-bit machines is essential for protecting many modern applications, but poses challenges over strictly 64-bit code because of the sparsity of available registers, the difficulty of identifying 32-bit PIC instructions (which are non-returning calls on x86, and therefore resemble function calls), and the constrained address space available for CFI data pages. Intel ISAs also raise challenges related to unaligned memory accesses. The overall approach generalizes to 64-bit code (and raises fewer implementation challenges on those ISAs).

Our prototype targets Linux applications and assumes that dynamic code generation uses POSIX memory allocation and protection functions. Non-standard memory APIs can be supported by adding their ABI signatures to the list of calls monitored by RENEW, and direct system calls can be supported by instrumenting `int 80` and `sysenter` instructions as system calls.

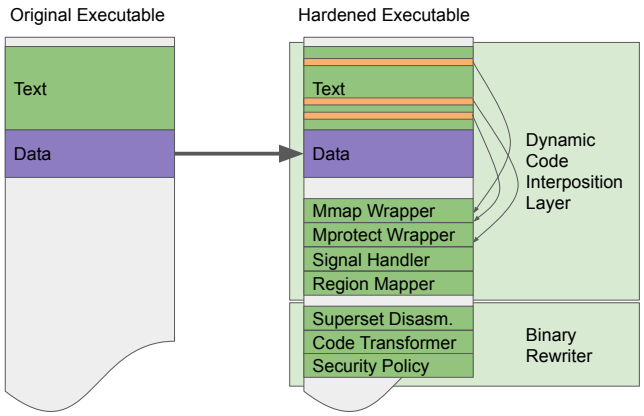

Figure 2: An overview of RENEW

2.5 Overview

Figure 2 depicts RENEW's architecture, comprised of two major components contained within the target application's address space: a dynamic code interposition layer (§3.1) and a binary rewriter (§3.2). The interposition layer contains memory protection hooks, a dynamic code region mapper, and a signal handler, while the rewriter contains a superset disassembler, code transformation component, and security policy component.

Before program execution, the interposition layer must be initialized. First, functions that change memory protections are hooked with wrapper functions. This prevents unsafe code from directly executing by intercepting attempts to set memory regions executable. By disallowing dynamically generated code to be set executable, the only dynamic code that runs is the code hardened by RENEW.

Our dynamic code region mapper tracks all dynamically generated code regions and maps them to rewritten versions. This tracks independent code regions of different sizes, and maps their addresses to instructions in rewritten code.

Our signal handler handles all segfaults. Since we prevent generated code from being set executable, this catches any attempt to run unsecured code, which we redirect to our rewritten, hardened version using our region mapping.

The rewriter's foundation is a superset disassembler, which disassembles from every offset in each generated code region, extracting all instruction sequences and eliminating duplicate and invalid sequences. This results in a *disassembly tree*, which encodes a superset of all reachable instruction sequences in the code page. The disassembly tree is passed to the code transformation component.

The code transformation component converts the original instructions to their rewritten versions while enforcing the security policy provided by the security policy component. Its transformations redirect control-flows, both to enforce policies and to ensure all rewritten jumps point to their corresponding rewritten targets. The code transformation also provides the basic security primitives needed to enforce security policies.

The security policy component defines legal jump targets for indirect control flows. This policy may be obtained by analyzing the generated code or can be a generic, static policy.

3 DETAILED DESIGN

3.1 Dynamic Code Interposition

RENEW securely redirects control and rewrites dynamically generated code by imposing a non-circumventable interposition layer that generically controls attempts to generate and execute code. The interposition layer uses the memory protection hooks to mediate attempts to change memory permissions, and adds dynamic code regions to our region maps. When the application attempts to execute dynamic code, the control transfer is redirected (e.g., by catching the segfault) to the corresponding rewritten code page using the region maps.

3.1.1 Memory Protection Hooks. Before the code is initially generated, an application must first allocate writable memory. Since virtual memory management is performed by the OS, the application must do so by performing a system call to request memory to be allocated by the kernel. We therefore achieve complete mediation of these system calls via a standard API hooking approach, which replaces the (non-writable) import address table (IAT) entries of kernel-provided memory management functions with user-mode wrapper functions that modify their behaviors. The untrusted application can only circumvent this interposition by first removing the write protections on IAT pages, or by first performing a policy-prohibited control-flow transfer (e.g., a syscall or a jump to a syscall within one of the wrapper functions). RENEW blocks both of these attacks recursively, since both require the application to have already broken out of the CFI sandbox in order to perform the prohibited permission elevation or control-flow transfer.

The most important of these API functions on Linux are `mmap` and `mprotect`. The `mmap` function maps an address range to the address space of a process. This memory may be backed by a file, or may be anonymous memory with no backing file. The calling code may specify a desired base address for the mapped memory, or may allow the kernel to choose the base address. The `mprotect` function changes the permissions of existing memory regions, which allows a program to set the code it has just generated in a writable region to be executable.

Both functions accept an argument that consists of protection bits, including readable (`PROT_READ`), writable (`PROT_WRITE`), and executable (`PROT_EXEC`). On modern ISAs, the protection bits are hardware-enforced; attempts to read from a non-readable page, write to a non-writable page, or transfer control to a non-executable page solicit a segfault, which the OS passes to the user process for handling if a signal handler is registered.

Figure 3 illustrates the logic of our `mmap` wrapper. In order to prevent a memory region from being writable and executable simultaneously, we mask off the `PROT_EXEC` permission if the `prot` argument has both `PROT_WRITE` and `PROT_EXEC` set. Once we have ensured the call to `mmap` does not contain W+X permissions, we fall through to the kernel-provided `mmap` function. If it succeeds, we record the address of the mapped region and mark it as unsafe (not rewritten). Rewriting the generated code page to harden it occurs later, when and if the application attempts to pass control to it.

Figure 4 illustrates the logic of our `mprotect` wrapper function. If `mprotect` is called with the `PROT_EXEC` permission, we remove that from the argument and check whether we already have a record

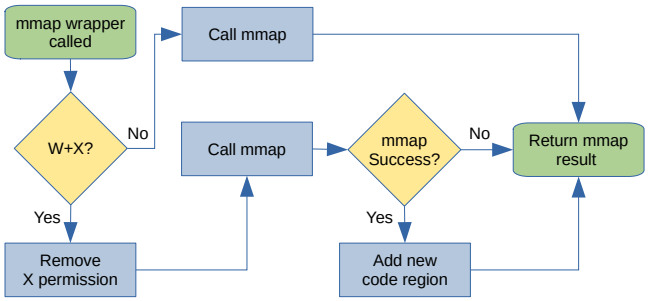

Figure 3: RENEW's `mmap` wrapper logic

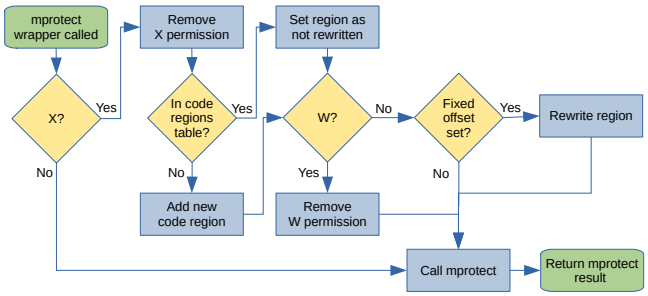

Figure 4: RENEW's `mprotect` wrapper logic

of it in our code regions table. If so, we mark that region as unsafe (not rewritten). Otherwise, we add a new region to the table. If the region is being set as W+X, then we also remove the `PROT_WRITE` permission to prevent the program from changing the original region undetected. If the region is being set executable but not writable, we check whether our fixed offset for rewritten code has been set. We set the fixed offset the first time the segfault handler is called to ensure that the initial memory layout of the application is known, including at least the first dynamic code region. If the fixed offset is set, then it is safe to rewrite the region immediately, so we harden it.

Memory regions that an application expects to be both writable and executable simultaneously are the most difficult (and dangerous) to support. Fortunately, most non-malicious applications do not actually require both permissions to be active on a single page at the same time. If they set both permissions, it is typically for expedience and efficiency (so that the permissions do not require later adjustment), but they subsequently perform writes on strictly non-executing pages, and they execute only pages that are not currently being modified. This is in part because writing to a currently executing page incurs an unnecessary performance penalty, since it impairs many hardware-assisted code optimizations, including speculative execution (which can less reliably anticipate which instructions might execute next) and caching (since writes can invalidate the instruction caches) [30].

In particular, we found that JIT compilers repeatedly modify dynamic code as they optimize, resulting in multiple generations of dynamic code that we must rewrite; but each generation only modifies future generations, not itself. Since these JIT compilers also allocate several separate regions of dynamic code, RENEW must ensure that all rewritten code can handle an update without breaking the code in a separate region. We handle this by treating direct jumps across regions as a special case, detailed in §3.2.

We conservatively prohibit writes to pages that are currently executing, since this behavior is the most rare (not exhibited by any non-malicious application we tested) and the most insecure. Such writes are prohibited by preventing any page from simultaneously having both write and execute permission bits set. Writes to currently executing pages could be supported by treating them as an (intercepted) write followed by an (also intercepted) implicit control transfer to the same page; however this is likely to introduce high overheads (since every write to the currently executing page would suffer a double-interception) and was not required to support any code behavior we encountered in non-malicious applications.

3.1.2 Dynamic Code Region Mapper. Programs that dynamically generate code often separate it into multiple non-contiguous memory regions. We must therefore track each generated code region to rewrite them and redirect control flows to the rewritten version of each.

From our tests, the number of code regions generated in even large applications tends to be small. We therefore maintain an array of region structs containing the addresses and sizes of original and rewritten regions, and a mapping between them. This array is not used by lookups in the rewritten code; it is only used by the static code when securing control-flow transfers to dynamic code.

The mapping component is called by our `mmap`/`mprotect` hooks (when a region is set executable), or by our signal handler (when the program attempts to execute a generated code region that is not yet rewritten). It in turn calls our binary rewriter, which generates both the rewritten version of the code and the mapping between the original and rewritten code. The mapping is an array of word-length offsets from the base address of the rewritten region, corresponding to each byte in the original code region. The mapping size is therefore the machine word size times the size of the original region. It is only used for the initial lookup when the signal handler is called and for generating the lookup table entries referenced by indirect control flow lookups within rewritten code.

The lookup table for each rewritten region is placed at a fixed offset from the original base address of a code region, and consists of an array of code pointers. The fixed offset is calculated the first time the rewriter is invoked so that the lookup table does not collide with any mapped regions. The rewritten indirect control-flow instructions use the fixed offset to perform lookups more efficiently (see §3.2).

3.1.3 Signal Handler. Securing control-transfers within the static code is the province of the static CFI or SFI enforcement mechanism. This mechanism should ideally hand control to RENEW when it detects a potential transfer from static code to data. However, to avoid any reliance on source code information in our evaluation of RENEW, we instead adopt the blunt approach of allowing such transfers to raise a segfault that is caught by RENEW's signal handler. This is slower than allowing CFI to implement a graceful control-flow transfer, but reflects the worst-case scenario that the CFI mechanism is unable to statically identify any of these transfers and must rely entirely upon the slow-path interception.

To implement this slow-path, we register a user-level segfault (SIGSEGV) handler, which is called whenever code attempts to jump

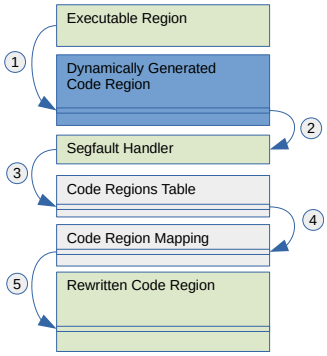

Figure 5: An example of the segfault handler intercepting an attempt to jump to a dynamically generated code region

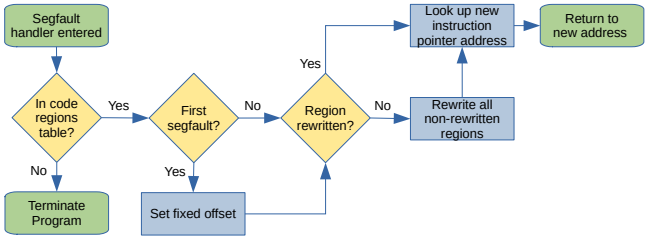

Figure 6: RENEW's segfault handler logic

to instructions in the dynamically generated regions that we have forced to be non-executable. Attacks that attempt to unregister, deprioritize, or replace the signal handler are blocked by enforcing a CFI policy that restricts calls to the signal handler system API.

Figure 6 shows the behavior of our segfault handler. First, it distinguishes valid code regions from invalid code regions by searching for the faulting address in the dynamic code regions table. Any address within one of our recorded regions is valid and can be translated to a rewritten region, while any other address is invalid (a program bug) and terminates the program. Since all valid addresses for RENEW are previously non-executable addresses that now contain dynamically generated code, the only possible sources of mistranslation by this procedure occur if the original program would have jumped to unmapped memory that is now occupied by a rewritten code page introduced by RENEW. In this case, the original program would have crashed, whereas the hardened program continues to execute (probably crashing soon after) in a way that is guaranteed to comply with the security policy.

If the address is found in our code regions table, then we check whether this is the first time the segfault handler has been invoked. If so, we calculate the fixed offset. By waiting until the program first attempts to execute a dynamic code region, we can use the base address of this region to calculate the fixed offset and avoid collisions of the fixed offset tables with any other memory regions.

The segfault handler then checks whether the region has been rewritten. If so, we can immediately look up the corresponding address in our rewritten code and set that to be the returning address from the segfault handler. If not, we must rewrite it first. Cross-region jumps might potentially jump into a region that we have not rewritten yet, triggering another segfault, so we avoid this by also rewriting all regions that we have in our code regions table that still need rewriting.

The handler returns to the new address, with all registers and the stack unchanged. The process is therefore transparent to the application, which continues to execute as if it had jumped directly to the rewritten code.

Figure 5 depicts the steps for intercepting a jump to dynamically generated code. **Step** ① shows code attempting to jump to a region that we prevented from being set as executable, and **Step** ② redirects control to our segfault handler. In **Step** ③, the handler finds the region in our code regions table. Since the region has already been rewritten, it gets a pointer to the mapping from the code regions table in **Step** ④, and it looks up and returns to the new address in **Step** ⑤.

Using the signal handler exclusively for handling attempts to jump to a dynamic code region imposes a performance penalty, as control being redirected to a signal handler is relatively slow compared to a direct jump. A significant performance improvement can be obtained by avoiding the segfault entirely, which can be achieved by inserting lookup calls before jumps in the code of the original program that are statically predictable as targeting dynamic code. This fast path is purely a performance optimization and is not needed for correctness or security. The optimization does not eliminate all slow-path cases, as callbacks in the original binary can return into dynamic code; but offers an avenue of incrementally improving performance through better static analysis. We experimented with this in Firefox's JIT compiler, manually inserting a single-line lookup in the `EnterJit` and `EnterBaseline` functions to obtain a minor performance improvement. Similarly, we added a single lookup in the `BC_JLOOP` assembly code in LuaJIT. Our evaluations of Spidermonkey and LuaJIT in §5 also contain this optimization.

3.2 Binary Rewriter

RENEW's binary rewriter is called by the dynamic code interposition layer to translate dynamically generated code to a safe version that is run instead of the original code. Because this must occur at runtime as code is generated, it must be lightweight and fast. The entire binary rewriter component resides in the address space of the host application.

3.2.1 Superset Disassembler. We use the superset disassembly technique described in MULTIVERSE [5], wherein every offset in a region is disassembled. The resulting instruction sequences are then trimmed to eliminate redundancy and stitched together.

Our disassembler must be small and fast. To achieve this, we use a disassembly algorithm that ignores security-irrelevant details about many instructions and their addressing modes. Detailed information is mainly needed for the subset of instructions that transfer control-flows; for most other instructions we only need to know the instruction's length. Some reassembly optimizations can also benefit from parsing arguments to additional instructions (e.g., constant-propagation for non-branching code that performs certain indirect calls). For these, the overhead of fully disassembling the relevant instructions is balanced against the observed performance benefit of the optimization.

Original	Rewritten

`jmp/call` *target*	*If target in same JIT region:*
	`push` ⟨*instruction address*⟩ *(if call)*
	`jmp` *newtarget*
	If target in different JIT region:
	`sub esp,0x4`
	`push` *target*
	`add esp,0x8`
	⟨*rewritten jmp/call*⟩ `[esp-0x4]`
	If target in non-JIT region:
	`push` ⟨*instruction address*⟩ *(if call)*
	`jmp` *target*

`jmp/call [`*m*`]`	`push eax`
	`mov eax, [`*m*`]`
	`test [eax*4+`⟨*fixed offset*⟩`], 0x3`
	`cmovne eax, [eax*4+`⟨*fixed offset*⟩`]`
	`and al, 0xf0`
	`push eax`
	`pop eax`
	`pop eax`
	`push` ⟨*instruction address*⟩ *(if call)*
	`jmp [esp-0x8]` *(0x4 if call)*

`ret (`*n*`)`	`xchg eax, [esp]`
	`test [eax*4+`⟨*fixed offset*⟩`], 0x3`
	`cmovne eax, [eax*4+`⟨*fixed offset*⟩`]`
	`xchg [esp], eax`
	`add esp,(`*n*`+)0x4`
	`and [esp-(`*n*`+)0x4], 0xf0`
	`jmp [esp-(`*n*`+)0x4]`

Figure 7: RENEW Code Transformations

Disassembly proceeds by starting at each offset and then performing fall-through disassembly until one of three conditions is met: (a) disassembly reaches the end of the code region, (b) an already-disassembled offset is encountered, or (c) a prohibited or invalid instruction is encountered. Disassembling starting from each offset generates instruction sequences that eventually result in an instruction address that is identical to an address already disassembled in a previous pass, and continuing to disassemble would produce duplicate entries. Therefore, when an already-encountered offset is found, the disassembler returns a special jump instruction that the code transformation component uses to stitch partially overlapping instruction sequences together.

Upon encountering a prohibited or invalid instruction, the disassembler indicates that the current instruction sequence is invalid, which the code transformation component can use to eliminate such sequences from the generated code.

3.2.2 Code Transformation. The code transformation component is the heart of RENEW. It transforms the original instructions returned by the superset disassembler into hardened instruction sequences with the primitives required for CFI and SFI policies.

CFI enforcement is bootstrapped by first dividing instructions into 16-byte aligned chunks and forcing indirect jumps to the start of chunks [41] by masking off indirect jump target addresses before the jump to restrict their destinations, and padding with `nop` instructions to ensure that no instruction spans a chunk boundary. This converts the superset disassembly into a series of uncircumventable basic blocks. Blocks ending in an indirect jump are then elaborated with CFI guard code that enforces an arbitrary control-flow policy.

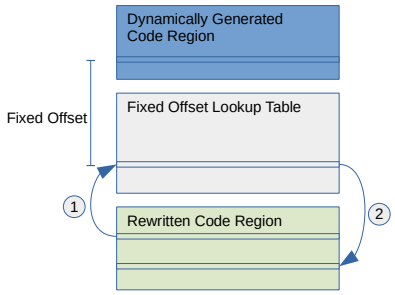

Figure 8: An example of rewritten code performing a lookup.

For speed, most instructions pass through the rewriter unmodified. The security-relevant instructions are control-flow transfers and instructions at identified jump target addresses. Instructions identified as jump targets are aligned to the start of a chunk (and preceded by a `nop` to mark them as targets), but all other instructions remain unchanged.

Figure 7 summarizes the code transformations implemented by RENEW. There are three major categories of control-flow instructions: direct control-flow instructions, indirect control-flow instructions, and return instructions (which are a special case of indirect control-flows). For simplicity, the figure omits the padding bytes (consisting of `nop` instructions) that must be inserted to ensure that none of the new instructions span adjacent chunks. The padding is carefully inserted to ensure that the masking instruction preceding a `jmp` is never placed in a separate chunk.

Direct control flow instructions include conditional jump instructions (such as `jne`) and direct call/jumps. Jumps within the same code region are the most straightforward to rewrite because the target offset is an immediate argument. When performing initial instruction transformations for each of these instructions, we generate a jump instruction with a placeholder offset and record a relocation entry, as the instruction may jump forward into not-yet-rewritten code. We also expand jump instructions with a short 1-byte offset to jumps with a word-sized offset instead, since our rewritten code expands in size. After initial rewriting, we then go through these relocation entries and update the offsets to jump to the new rewritten target. If an instruction is a call, then we must convert the call into a jump and insert an extra push instruction so that we push the old return address that the original call would have pushed on the stack. This is necessary for PIC code that uses return addresses to calculate offsets to data.

Cross-region direct jumps or calls are more complex. Were we to hard-code the addresses of rewritten jumps, then a jump to a region that is later modified could break. Even a small modification can change many target offsets due to superset disassembly's conservative retention of code blocks that it cannot prove are unreachable. While we could rewrite every region every time any region is modified, this would lead to significant rewriting overhead.

We therefore allow separate regions to be updated independently by transforming cross-region direct control-flow instructions into indirect control-flow instructions, yielding the redirected control-flow illustrated in Figure 8. A lookup for the cross-region jump is hence performed at the moment the jump is taken, ensuring that cross-region jump targets are never stale.

Loop instructions (`loop`, `loope`, and `loopne`) are a special case of conditional control flow instructions. They decrement `ecx` without changing status flags. They also only have a single-byte offset, which we must expand without corrupting flags. We do this by using a loop instruction that targets a jump placed immediately before it, which jumps to the target of the original loop instruction.

Indirect control-flow instructions require insertion of extra instructions, as it is impossible to reliably predict their destinations until runtime. We therefore insert instructions that dynamically lookup the new rewritten target address from the original target address. Speed is essential, as indirect transfers are frequent in most binaries. Therefore, we store the target address in a lookup table at a fixed offset from the original target, which we calculate the first time the rewriter is invoked to avoid memory conflicts. The instructions perform an add (`original address + offset`) and read the bytes at that address. We use the lower bits to mark whether an offset is a valid target to distinguish stale addresses (which need redirection) from rewritten or external static addresses (which must not be redirected). We do not rely on these bits for security, as we always clear them from the address before jumping. This ensures that even if an attacker can control the target address, the control-flow is sandboxed (chunk-aligned) regardless of whether a lookup is performed. Our lookup table includes a distinct entry for every address in the original code.

The original target of an original `jmp/call` instruction is encoded in its ModR/M byte, optional SIB byte, and optional displacement bytes. We transplant these bytes into a `mov` instruction (line 13 of Figure 7) so that we can extract the target address for lookup. Since our rewritten code saves the value of `eax` on the stack and thus temporarily changes `esp`, we must treat instructions that refer to `esp` (e.g., `jmp [esp]`) as a special case. We therefore add a word-sized displacement to the original memory encoding, or expand the existing displacement to compensate for the changed stack pointer.

Figure 8 shows a simplified example of the steps in a rewritten indirect control flow performing a lookup. In **Step** ① the rewritten instruction adds the fixed offset to the original address to read the new destination address. **Step** ② masks the address to be 16-byte aligned and jumps to it.

Return instructions must be rewritten to retrieve the return address from the top of the stack, look it up similarly to indirect control-flow instructions, mask off the resulting address, and jump to it. The process, while similar to other indirect control-flow instructions, requires some slightly different code. Extra instructions are required for return instructions that take an immediate value to adjust the stack pointer, since the appropriate adjustment must then be computed at runtime.

3.2.3 Security Policy Enforcement. RENEW's policy enforcement comes after each jump target lookup; the lookup results are not trusted. Attacker corruption of lookup table data and other code pointers therefore does not suffice to circumvent the sandboxing guards. RENEW clears low-order bits of all target addresses after looking them up to chunk-align them. Since no instructions span a chunk boundary in rewritten code, this masking prevents any indirect jump from targeting an instruction sequence we do not control. The start of each chunk is also the start of a rewritten instruction, and therefore any attempt to escape our sandbox by controlling indirect targets is thwarted. Additional policies can be enforced by performing additional checks on addresses, and security enforcement comes from validating whether the destination address is permitted by each policy.

To perform our evaluation, we empirically derived and enforced a CFG policy for each tested application. The policies were derived by tracing the original applications (e.g., with JIT enabled but dynamic code left unsecured) on a variety of benign inputs to learn recognizable byte patterns at targets of indirect jumps. This results in many instructions being incorrectly identified as jump targets, expanding rewritten code size and lowering performance, but not affecting security. More precise policies can be derived by applying prior research on policy inference for control-flow sandboxing [22, 31–33].

4 IMPLEMENTATION

RENEW comprises about 2,000 lines of C code with some in-lined assembly and raw machine code sequences used in instruction translation. Its small size is the result of our space-optimizing design, since both of RENEW's major components—the dynamic code interposition layer and the binary rewriter—must be placed into the target application being protected and must therefore avoid high space overheads.

RENEW can be added to target applications by modifying the application binary or by linking it in during compilation. For our experiments we chose the latter approach, statically linking RENEW in the host program by passing the appropriate compiler flags (`-Wl,-wrap=mmap -Wl,-wrap=mprotect`) to the linker in order to wrap the memory-management API functions, and manually inserting a single call in the `main` function to register the segfault handler. For our integration with UPX, we include the RENEW binary as a shared library, except that we load it manually with a small assembly routine, and the memory management API functions in the UPX decompression routines were wrapped manually since UPX uses a non-standard IAT.

Our disassembler is embedded in the process's address space so that it can disassemble instructions dynamically at runtime. Disassembling binary code requires identifying all execution paths and decoding each instruction along each path. To achieve path identification, we implemented a superset disassembler in the style of MULTIVERSE [5], but adapted as described in §3.2.1 to disassemble code pages in running process images rather than file system executable binaries. The instruction decoding subtask was performed by the udis86 library [61], which was chosen for its small size and lack of dynamic memory allocation. We made minor additions and fixes to handle a few instructions that udis86 incorrectly decoded.

5 EVALUATION

In order to demonstrate RENEW's applicability for diverse forms of dynamically generated code, we tested two distinct types: JIT compilers and binary packers. Our modified LuaJIT is forked from commit `3f9389` from June 2021, our modified Firefox is based on Firefox's version 83.0 release, and our modified UPX is forked from commit `66fe8a` from May 2021. Our test machine runs Ubuntu 20.04.1 LTS with an Intel i7-2600 CPU running at 3.40GHz, with 12GB of RAM.

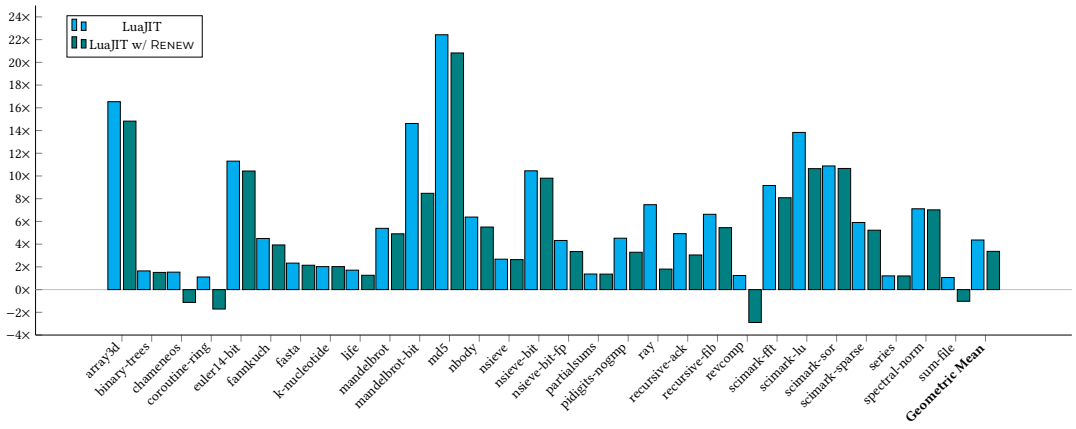

Figure 9: Overhead of Renew on Lua benchmarks. Values represent benchmark speed with JIT+Renew relative to JIT disabled. Higher is better, and negative values mean the benchmark ran slower than with JIT disabled.

Table 1: Runtime performance in ms of LuaJIT with and without Renew (lower is better), with the percentage JIT speed-up preserved by Renew (higher is better)

Benchmark	No JIT	Unsecured JIT	JIT + Renew
array3d	4300	260	290 (99%)
binary-trees	3600	2190	2380 (87%)
chameneos	3500	2280	3950 (–)
coroutine-ring	900	810	1540 (–)
euler14-bit	12210	1080	1170 (99%)
fannkuch	20200	4490	5140 (96%)
fasta	16060	6890	7490 (93%)
k-nucleotide	9100	4500	4500 (100%)
life	770	450	610 (50%)
mandelbrot	13750	2550	2800 (98%)
mandelbrot-bit	25000	1710	2950 (95%)
md5	26240	1170	1260 (100%)
nbody	8810	1380	1600 (97%)
nsieve	5650	2110	2140 (99%)
nsieve-bit	11080	1060	1130 (99%)
nsieve-bit-fp	10980	2540	3280 (91%)
partialsums	2590	1890	1900 (99%)
pidigits-nogmp	11670	2580	3550 (89%)
ray	10760	1440	5960 (52%)
recursive-ack	640	130	210 (84%)
recursive-fib	6760	1020	1240 (96%)
revcomp	1240	1000	3590 (–)
scimark-fft	37750	4120	4670 (98%)
scimark-lu	31400	2270	2950 (98%)
scimark-sor	21110	1940	1980 (100%)
scimark-sparse	20440	3460	3910 (98%)
series	2330	1930	1940 (100%)
spectral-norm	15720	2210	2240 (–)
sum-file	680	640	700 (91%)
Geometric Mean	6839	1567	2031 (77%)

5.1 JIT Compilers

We evaluated Renew on two of the most widely used JIT compilers: LuaJIT [52] for Lua, and Spidermonkey [43] (Firefox's JIT compiler) for JavaScript. Both of these JIT compilers generate native code from scripting language code at runtime, but their code generation approaches are highly dissimilar due to the significant differences in the two scripting languages they target. This makes them challenging tests of source-agnostic, binary-level CFI. Both JITs are also unsupported by all existing CFI-JIT solutions [4, 48, 75], which remain incompatible with all major JIT implementations from the past 8+ years.

We inserted Renew into both and ran the resulting binaries on test suites and benchmarks for Lua and JavaScript, respectively. Since dynamic code generation by JIT compilers is primarily motivated by performance, but security-sensitive users must presently disable the JIT to attain CFI-level assurances, the goal of Renew is to achieve a runtime overhead that is lower than disabling the JIT (but probably higher than a JIT without any CFI security, which constitutes a lower bound on the attainable performance overhead). Comparison with prior JIT-CFI solutions [4, 48, 75] is impossible since none of those solutions work on modern versions of LuaJIT and Spidermonkey (i.e., they crash all the tests).

For Lua, we therefore evaluated the performance of the set of benchmarks used by the LuaJIT team to compare LuaJIT with the official Lua interpreter, which are 29 Lua scripts that perform a variety of algorithms and common scientific computing tasks [37, 38]. We ran the benchmarks with the same arguments as the LuaJIT team, running the LuaJIT interpreter with JIT-compilation disabled (via the `-joff` flag), with JIT-compilation enabled, and with JIT-compilation enabled with Renew. All benchmarks run identically with and without Renew, showing the high compatibility of our approach. We followed the precedent of the LuaJIT team, running each benchmark three times and selecting the fastest result.

Table 1 lists performance results for LuaJIT with JIT disabled (`-joff`), LuaJIT, and LuaJIT with Renew, with the final column listing the percentage of JIT performance improvement that Renew was able to preserve. Figure 9 compares Renew's performance to LuaJIT with JIT disabled, showing how many times faster (or slower, for negative values) a benchmark runs relative to the JIT-disabled interpreter.

On average, using Renew to secure the JIT code exhibits a 3.4× performance improvement over disabling the JIT to achieve equivalent security. Using the JIT compiler without any CFI security is 4.4× faster than leaving it off, meaning that Renew retains 77% of the performance advantages of LuaJIT compilation, but without the associated security risks.

Table 2: Octane results in Spidermonkey (higher is better)

Benchmark	No JIT	Unsecured JIT	JIT + RENEW
Richards	111.00	6166.33	2842.00
DeltaBlue	116.33	7140.67	866.00
Crypto	250.67	8594.33	5127.33
RayTrace	259.67	13150.00	3363.67
EarleyBoyer	410.00	16645.00	7168.00
RegExp	171.67	5322.00	2092.67
Splay	628.33	4869.33	3376.67
SplayLatency	3293.67	4632.33	841.00
NavierStokes	409.00	18803.00	14672.67
PdfJS	1434.00	15010.67	688.00
Mandreel	125.67	10153.00	2534.33
MandreelLatency	784.00	16444.33	3937.67
Gameboy	1028.67	41966.33	3264.67
CodeLoad	15171.00	15606.33	4919.00
Box2D	563.33	18634.00	429.00
zlib	45196.00	45321.33	30314.67
Typescript	2243.67	19491.33	1665.33
Geometric Mean	742.33	12590.00	2865.00

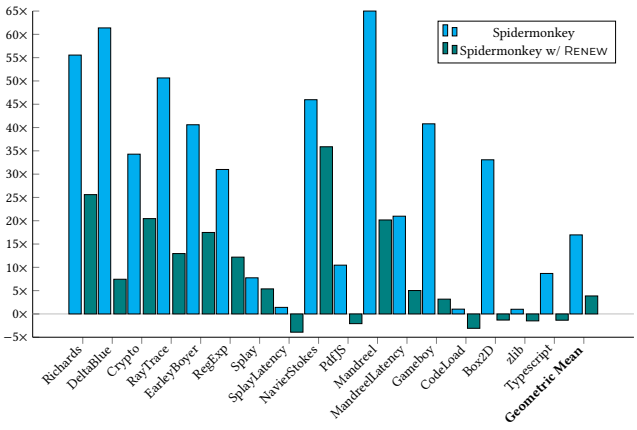

Figure 10: Overhead of RENEW on Octane benchmarks. Values represent benchmark speed with JIT+RENEW relative to JIT disabled (`-no-blinterp`). Higher is better, and negative values indicate a slowdown.

The four benchmarks that exhibited worse performance than disabling the JIT yield only marginal speed increases by enabling the original JIT, suggesting that in these cases dynamic code is ineffective for improving performance. For example, `sum-file` consists of only a simple loop that sums numbers from an input file. The overhead from our rewritten JIT code eclipses the minor performance improvement from JIT-compiling this single addition within the loop. Two other benchmarks (`chameneos`, `coroutine-ring`) make extensive use of Lua coroutines, which LuaJIT likewise struggles to optimize, leaving little performance for RENEW to preserve.

Rewriting the LuaJIT binary with RENEW increases its size from 542KB to 676KB (25%). This is a fixed file size overhead that does not depend on the sizes of the Lua bytecode binaries that LuaJIT compiles (and whose sizes remain unchanged since RENEW does not statically modify them).

We also evaluated RENEW on Spidermonkey, the JavaScript engine at the core of Firefox. Since 2019, Spidermonkey has had three tiers of dynamic code generation [15]: a Baseline Interpreter, which

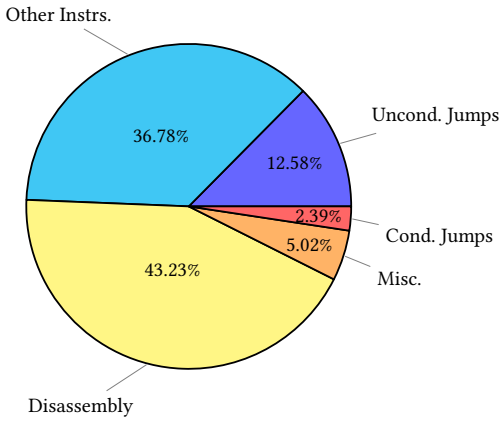

Figure 11: RENEW overhead breakdown in Octane

is a JavaScript interpreter generated on the fly; a baseline JIT compiler to quickly generate machine code for JavaScript bytecode instructions; and Ion (replaced by the newer Warp in 2020 [16]), an optimizing JIT compiler that does heavier analysis for optimized code generation. Spidermonkey can be run with no dynamic code generation by disabling the Baseline Interpreter, which then only uses the base C++ interpreter.

We tested RENEW on Firefox's JIT compiler test suite, which consists of 6206 relevant tests. JIT+RENEW passes all tests, demonstrating RENEW's high compatibility. Rewriting with RENEW increases the size of the Spidermonkey binary by less than 1% because Spidermonkey is 460MB (significantly larger than LuaJIT) and RENEW supplements it with a small, fixed library of size 0.53MB.

Figure 10 shows the performance results for the Octane JavaScript benchmark suite v9. We compare the performance improvement of running the unmodified JIT compiler with running only the C++ interpreter (with the Baseline Interpreter disabled via the `-no-blinterp` flag). Octane provides an overall score by taking the geometric mean of the individual benchmarks. We ran the entire benchmark three times and took the arithmetic mean of the individual and overall scores.

Table 2 reports the results. On average, securing the JIT code dynamically with RENEW is 3.9× faster than disabling JIT compilation to enforce static CFI. Spidermonkey's more dramatic impact on JavaScript performance (17.0× faster than without JIT compilation) and Octane's narrower focus on JIT-impacting performance metrics means that RENEW preserves 23% of the performance advantages attainable by unsecured JIT code on Firefox as measured by Octane.

As with the Lua benchmarks, a few benchmarks are significant outliers with substantial drops in performance under RENEW, and the performance loss is specifically associated with benchmarks that exhibit little or no performance benefit from JIT compilation even without CFI security. In these cases the benefit of JIT compilation is not enough to offset the overhead of securing the dynamically generated code. One way to improve performance of these outliers is to conditionally disable the JIT when the JIT's profiling indicates that dynamic code is not benefiting performance, as recommended by prior JIT optimization research [49].

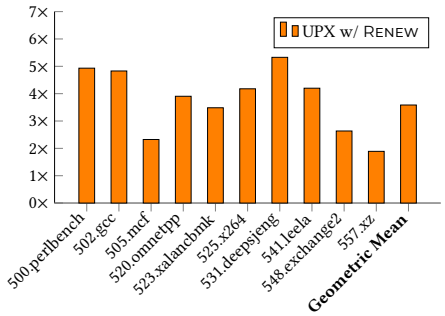

Figure 12: Overhead of UPX with Renew on SPEC CPU 2017 Integer benchmarks. Values represent benchmark speed with UPX+Renew relative to unmodified UPX.

Figure 11 breaks down the sources of Renew's overhead on the Octane benchmarks. The majority of the overhead time is devoted to rewriting the JIT code: Processing and aligning the large number of security-insensitive, non-control-flow instructions in dynamic code regions consumes 36.78% of the time. A further 12.58% is spent rewriting unconditional control flow instructions, including call instructions, and 2.39% of overhead is spent on rewriting conditional jumps. Another large percentage of the overhead (43.23%) consists of performing superset disassembly on the JIT code. The miscellaneous category in the figure includes segfault handling, rewriting return instructions, and overhead from the `mmap` and `mprotect` wrappers, all of which contribute only a small portion of the overhead. This implies that future advances in dynamic code hardening will benefit most from advances in rewriting and disassembly efficiency.

5.2 Executable Packers

UPX [51] is used to compress large executables, which self-unpack at runtime. Passing UPX an uncompressed `linux/elf386` binary generates a compressed binary with a small uncompressed stub that unpacks the compressed payload in two stages. The tiny stub unpacks more complex decompression routines, which then unpack the entire binary image in memory. Then the UPX code unmaps itself and returns control to the original binary code.

We modified the UPX binary so that the compressed binaries it produces load Renew as a dynamic library prior to extraction. In the uncompressed initialization code, we inserted a call to load and initialize a relocatable Renew blob. The `mmap` and `mprotect` calls in the stub and in the second decompression stage are instrumented with wrappers to prevent them from setting any regions it maps as executable. We rewrite a region when an `mprotect` call attempts to set it to executable. When the stub jumps into decompressed code, our `sigsegv` handler catches the segfault and redirects the control to our rewritten version.

We ran our modified UPX with its default compression settings and tested it on 14 GNU binutils 2.37 binaries, including `ar`, `strip`, `objdump`, `readelf`, and `ld`. All binaries worked with Renew without any issues, except that the `as` assembler is incompatible because it attempts to register a segfault handler that conflicts with Renew's handler. This compatibility issue could be addressed in future work as described in §2.4. Renew has almost no impact on compressed binary size (less than 0.01% increase), making it effective for preserving UPX's space-efficiency while protecting against its security risks. This is because Renew's initialization code only adds a constant 436 bytes to each compressed binary. The average compression ratio for our Renew-UPX compressed binaries was 37.745%, less than a tenth of a percent larger than the unmodified UPX's 37.735% compression ratio.

We also tested our modified UPX with the 10 SPEC CPU Integer benchmarks to test performance. Since the entire compressed binary is extracted before the program starts, Renew rewrites the entire binary before it begins to execute. This results in a brief pause before the program runs and some slowdown during execution on the order of 3.6× slower than the original SPEC benchmarks. This demonstrates that our approach can be applied to programs that dynamically generate native code, yet are vastly different from JIT compilers.

5.3 Example Security Policy

To test Renew in a practical policy enforcement, we implemented an overwriting, no zeroing parallel shadow stack [14]. Renew adds the shadow stack to UPX-compressed binaries dynamically, right after they decompress. This contributes an extra `mov` instruction before each rewritten call, and extra `add` and `push` instructions before each rewritten return. The enforced CFI policy protects the shadow stack's integrity and requires callee returns to target valid call sites in the current call chain. Forward edges are not constrained by the shadow stack; they are constrained by the trace-learned CFI policy described in §3.2.3.

Evaluation using the SPEC CPU Integer benchmarks shows that the hardened code passes all tests in the provided test dataset, and exhibits a median 2% overhead relative to no security policy. We emphasize that these results are exhibited with no application-specific customization to Renew and no manual changes to any of the compressed applications.

6 RELATED WORK

Binary Rewriting. Binary rewriting is an active area of research that dates back more than 50 years [17]. The binary rewriting works most relevant to Renew concern the challenge of adding CFI instrumentation to binary code.

Dynamic binary rewriters such as DynamoRIO [8], Valgrind [45], and DynInst [6] instrument dynamically generated code through dynamic interpretation, but incur high overheads due to context switches between target code and interpreter. Static binary rewriters, such as Uroboros [69] and Ramblr [67], quickly disassemble and reassemble binaries with instrumentation, but cannot support dynamic code.

Multiverse [5] addresses the problem of obtaining a static disassembly approximation that is guaranteed to conservatively include all reachable code. This strategy is leveraged and adapted to dynamic code by Renew.

Control Flow Integrity. Since the advent of CFI [1], it has seen active development in both offensive and defensive directions. A 2018 survey measured the performance of around 25 prominent CFI tools [10]. Major innovations include applications of CFI to COTS binaries on Linux [77] and Windows [71], CFI enhancements

Table 3: Comparison of RENEW with related defenses

System	Year	SFI	CFI	Code-reuse Immunity	JIT	Packers	Source-agnostic
NaCl-JIT	2011	✓	✗	✓	✓	✗	✗
Librando	2013	✗	✗	⚠	✓	✗	✓
RockJIT	2014	✓	✓	✓	✓	✗	✗
SDCG	2015	✗	✗	✗	✓	✗	✗
JITScope	2015	✓	✓	✓	✓	✗	✗
JITGuard	2017	✗	✗	⚠	✓	✗	✗
RENEW	2022	✓	✓	✓	✓	✓	✓

⚠ = Defense is diversity-based, so can be compromised by information disclosure.

for on-site randomization and progressive deployment [76], finer-grained policy enforcement [23, 28], cryptographic protections [40], opacity against implementation disclosure attacks [42], improved modularity [47], and controls for object-oriented control-flows [70]. BPA [33] provides a mechanism to help build better CFI policies. ARCUS [74] can be deployed alongside CFI to find the root cause of a policy violation. OS-CFI [32] and CFI-LB [31] use runtime data to enforce dynamic policies. However, none of these works support dynamically generated code.

Recent research has also sought to address increasingly dangerous attacks that exploit policy loopholes or abuse imprecision introduced by some CFI controls to improve efficiency [12, 20, 24, 29, 39, 58]. To identify such weaknesses, ConFIRM [73] and CFIBench [35] analyze the compatibility and security of CFI implementations, and LLVM-CFI [44] analyzes the effectiveness of CFI policies.

JIT Defenses. The security risks engendered by JIT-compiled code have been recognized for decades (cf. [25]), giving rise to a history of defensive efforts.

NaCl-JIT [4] leverages Native Client to sandbox two JIT compilers (the V8 JavaScript engine and the Mono C# framework) and their generated code using SFI. This provides less robust protection than a CFI solution and requires porting the JIT compilers to Native Client. Secure Dynamic Code Generation (SDCG) [60] enforces W⊕X permissions on JIT code by moving the code generation for a JIT engine (V8 and the Strata dynamic translation engine) to a separate trusted process with write permissions, while leaving the executable code as read-only in the main process. This is effective against code injection attacks but not code-reuse attacks.

Librando [27] randomizes the output of the V8 and HotSpot JIT compilers by intercepting and randomizing dynamically generated code. It uses a custom segfault handler in a manner similar to RENEW to intercept attempts to modify or jump to dynamically generated code. JITGuard [21] takes a different approach by using SGX to isolate and randomize code generated by Spidermonkey's JIT compiler. Although this approach requires special SGX hardware features, it also protects against data-only attacks. However, randomization-based defenses have the disadvantage of being potentially vulnerable to information disclosure attacks [19, 54, 59].

RockJIT [48] and JITScope [75] add coarse-grained CFI protection to V8 and fine-grained CFI protection to Spidermonkey, respectively. Both approaches do so by modifying the JIT compiler source code to implement source-aware CFI code generation technologies. While this constitutes an efficient and robust solution for those particular applications, it does not generalize to other forms of dynamically self-modifying code, and it contributes an additional layer of complexity to the application design that must be carefully maintained across implementation changes in order to avoid the aforementioned subtle CFI security lapses raised by the offensive security literature.

Table 3 summarizes the capabilities and compatibility characteristics of the most related prior works and RENEW. In general, none of the prior CFI or SFI works address non-JIT forms of dynamic code generation (e.g., self-unpacking executables), and most require information derived from program source code (e.g., scripting language sources) to effectively generate or modify CFI-protected binary code.

7 CONCLUSION

RENEW extends strong CFI and SFI protections to the growing class of software that includes dynamic code generation. By leveraging recent advances in rapid disassembly-reassembly, RENEW rewrites and sandboxes dynamically generated code as the untrusted application executes, achieving a 3–4× performance improvement over disabling dynamic code generation in JIT compilers to attain equivalent security. The approach also exhibits high robustness, preserving the behavior of Lua, Firefox, UPX, and binutils across thousands of tests with little or no customization to each new application. A combination of fast-path and slow-path interception provides opportunities for future performance improvements through better static code analysis without sacrifices to security.

8 AVAILABILITY

RENEW sources are publicly available on GitHub at the following URL: https://github.com/SoftwareLanguagesSecurityLab/RenewCFI

ACKNOWLEDGMENTS

The research presented herein was supported in part by ONR award N00014-21-1-2654, DARPA award N6600121C4024, ARO award W911NF2110081, and an endowment from the Louis A. Beecherl, Jr. family. Any conclusions, recommendations, or opinions expressed are those of the authors and not necessarily of the above supporters.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*. 340–353.
[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information and System Security (TISSEC)* 13, 1 (2009), 4:1–40.
[3] Ittai Anati and Oren Ben Simhon. 2017. Control Flow Enforcement Technology (CET). Compiler Architecture and Tools Conference (CATC).
[4] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff L. Biffle, and Bennet Yee. 2011. Language-independent Sandboxing of Just-in-time Compilation and Self-modifying Code. In *Proceedings of the 32nd ACM Conference on Programming Language Design and Implementation (PLDI)*. 355–366.
[5] Erick Bauman, Zhiqiang Lin, and Kevin W Hamlen. 2018. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics.. In *Proceedings of the 25th Annual Network & Distributed System Security Symposium (NDSS)*.
[6] Andrew R. Bernat and Barton P. Miller. 2011. Anywhere, Any-time Binary Instrumentation. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools (PASTE)*. 9–16.
[7] Dionysus Blazakis. 2010. Interpreter Exploitation. In *Proceedings of the 4th USENIX Workshop on Offensive Technologies (WOOT)*.

[8] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An Infrastructure for Adaptive Dynamic Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. 265–275.

[9] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. 2011. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*. 463–469.

[10] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2018. Control-flow Integrity: Precision, Security, and Performance. *Comput. Surveys* 50, 1 (2018).

[11] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs.. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 209–224.

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

[13] Fred Cohen. 1986. *Computer Viruses*. Ph. D. Dissertation. U. Southern California.

[14] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. 2015. The Performance Cost of Shadow Stacks and Stack Canaries. In *Proceedings of the ACM Asia Conference on Information, Computer and Communications Security (AsiaCCS)*. 555–566.

[15] Jan de Mooij. 2019. The Baseline Interpreter: A Faster JS Interpreter in Firefox 70. *Mozilla Hacks* (2019). https://hacks.mozilla.org/2019/08/the-baseline-interpreter-a-faster-js-interpreter-in-firefox-70.

[16] Jan de Mooij. 2020. Warp: Improved JS Performance in Firefox 83. *Mozilla Hacks* (2020). https://hacks.mozilla.org/2020/11/warp-improved-js-performance-in-firefox-83.

[17] Peter Deutsch and Charles A. Grant. 1971. A Flexible Measurement Tool for Software Systems. In *Proceedings of the IFIP Congress, Volume 1*. 320–326.

[18] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. 2019. Leopard: Identifying Vulnerable Code for Vulnerability Assessment Through Program Metrics. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE)*. 60–71.

[19] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard E. Shrobe, Stelios Sidiroglou-Douskos, Martin C. Rinard, and Hamed Okhravi. 2015. Missing the Point(er): On the Effectiveness of Code Pointer Integrity. In *Proceedings of the 36th IEEE Symposium on Security & Privacy (S&P)*. 781–796.

[20] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard E. Shrobe, Martin C. Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*. 901–913.

[21] Tommaso Frassetto, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. JITGuard: Hardening Just-in-time Compilers with SGX. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. 2405–2419.

[22] Debin Gao, Michael K. Reiter, and Dawn Song. 2004. Gray-box Extraction of Execution Graphs for Anomaly Detection. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*. 318–329.

[23] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. 2016. Fine-grained Control-flow Integrity for Kernel Software. In *Proceedings of the 37th IEEE Symposium on Security & Privacy (S&P)*. 179–194.

[24] Enes Göktaş, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. 2014. Out of Control: Overcoming Control-flow Integrity. In *Proceedings of the 35th IEEE Symposium on Security & Privacy (S&P)*. 575–589.

[25] Willem De Groef, Nick Nikiforakis, Yves Younan, and Frank Piessens. 2010. JITSec: Just-in-time Security for Code Injection Attacks. In *Proceedings of the 5th Benelux Workshop on Information and System Security (WISSEC)*.

[26] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. 2006. Computability Classes for Enforcement Mechanisms. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28, 1 (2006), 175–205.

[27] Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. 2013. Librando: Transparent Code Randomization for Just-in-time Compilers. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*. 993–1004.

[28] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. 2018. Enforcing Unique Code Target Property for Control-flow Integrity. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*. 1470–1486.

[29] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-oriented Programming: On the Expressiveness of Non-control Data Attacks. In *Proceedings of the 37th IEEE Symposium on Security & Privacy (S&P)*. 969–986.

[30] Intel®. 2023. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel® Corporation, Chapter 3.6.8: Mixing Code and Data.

[31] Mustakimur Khandaker, Abu Naser, Wenqing Liu, Zhi Wang, Yajin Zhou, and Yueqiang Cheng. 2019. Adaptive Call-site Sensitive Control Flow Integrity. In *Proceedings of the 4th IEEE European Symposium on Security & Privacy (EuroS&P)*. 95–110.

[32] Mustakimur Rahman Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. 2019. Origin-sensitive Control Flow Integrity. In *Proceedings of the 28th USENIX Security Symposium*. 195–211.

[33] Sun Hyoung Kim, Cong Sun, Dongrui Zeng, and Gang Tan. 2021. Refining Indirect Call Targets at the Binary Level. In *Proceedings of the 28th Annual Network & Distributed System Security Symposium (NDSS)*.

[34] Johannes Kinder and Helmut Veith. 2008. Jakstab: A Static Analysis Platform for Binaries. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV)*. 423–427.

[35] Yuan Li, Mingzhe Wang, Chao Zhang, Xingman Chen, Songtao Yang, and Ying Liu. 2020. Finding Cracks in Shields: On the Security of Control Flow Integrity Mechanisms. In *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*. 1821–1835.

[36] Jay Ligatti, Lujo Bauer, and David Walker. 2005. Edit Automata: Enforcement Mechanisms for Run-time Security Policies. *International Journal of Information Security* 4, 1–2 (2005), 2–16.

[37] LuaJIT. Accessed 2022-05-02. Cleanup Workspace for LuaJIT Tests. https://github.com/LuaJIT/LuaJIT-test-cleanup.

[38] LuaJIT. Accessed 2022-05-02. Performance Comparison. https://luajit.org/performance.html.

[39] Andrea Mambretti, Alexandra Sandulescu, Alessandro Sorniotti, William Robertson, Engin Kirda, and Anil Kurmus. 2021. Bypassing Memory Safety Mechanisms Through Speculative Control Flow Hijacks. In *Proceedings of the 42nd IEEE Symposium on Security & Privacy (S&P)*. 633–649.

[40] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. 2015. CCFI: Cryptographically Enforced Control Flow Integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*. 941–951.

[41] Stephen McCamant and Greg Morrisett. 2006. Evaluating SFI for a CISC Architecture. In *Proceedings of the 15th USENIX Security Symposium*.

[42] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W. Hamlen, and Michael Franz. 2015. Opaque Control-flow Integrity. In *Proceedings of the 22nd Annual Network & Distributed System Security Symposium (NDSS)*.

[43] Mozilla Foundation. 2019. SpiderMonkey: The Mozilla JavaScript Runtime. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey.

[44] Paul Muntean, Matthias Neumayer, Zhiqiang Lin, Gang Tan, Jens Grossklags, and Claudia Eckert. 2019. Analyzing Control Flow Integrity with LLVM-CFI. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*. 584–597.

[45] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. *ACM Sigplan Notices* 42, 6 (2007), 89–100.

[46] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. 2007. Predicting Vulnerable Software Components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*. 529–540.

[47] Ben Niu and Gang Tan. 2014. Modular Control-flow Integrity. *SIGPLAN Notices* 49, 6 (2014), 577–587.

[48] Ben Niu and Gang Tan. 2014. RockJIT: Securing Just-in-time Compilation Using Modular Control-flow Integrity. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*. 1317–1328.

[49] Johnathan Norman. 2021. Super Duper Secure Mode. https://microsoftedge.github.io/edgevr/posts/Super-Duper-Secure-Mode.

[50] Johnathan Norman. 2022. Introducing Enhanced Security for Microsoft Edge. https://microsoftedge.github.io/edgevr/posts/Introducing-Enhanced-Security-for-Microsoft-Edge.

[51] Markus F.X.J. Oberhumer, László Molnár, and John F. Reiser. 2018. UPX: the Ultimate Packer for eXecutables. http://upx.sourceforge.net.

[52] Mike Pall. 2019. The LuaJIT Project. https://luajit.org.

[53] Tom Ritter. 2021. Browser Exploit History. Mozilla. docs.google.com/spreadsheets/d/1FslzTx4b7sKZK4BR-DpO45JZNB1QZF9wuijK3OxBwr0.

[54] Robert Rudd, Richard Skowyra, David Bigelow, Veer Dedhia, Thomas Hobson, Stephen Crane, Christopher Liebchen, Per Larsen, Lucas Davi, Michael Franz, Ahmad-Reza Sadeghi, and Hamed Okhravi. 2017. Address Oblivious Code Reuse: On the Effectiveness of Leakage Resilient Diversity. In *Proceedings of the 24th Annual Network & Distributed System Security Symposium (NDSS)*.

[55] Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. 2014. Predicting Vulnerable Software Components Via Text Mining. *IEEE Transactions on Software Engineering (TSE)* 40, 10 (2014), 993–1006.

[56] Fred B. Schneider. 2000. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)* 3, 1 (2000), 30—-50.

[57] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. 2017. Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis? *Comput. Surveys* 49, 1 (2017).

[58] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proceedings of the 36th IEEE Symposium on Security & Privacy (S&P)*.

[59] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-in-time Code Reuse: On the Effectiveness of Fine-grained Address Space Layout Randomization. In *Proceedings of the 34th IEEE Symposium on Security & Privacy (S&P)*. 574–588.

[60] Chengyu Song, Chao Zhang, Tielei Wang, Wenke Lee, and David Melski. 2015. Exploiting and Protecting Dynamic Code Generation. In *Proceedings of the 22nd Annual Network & Distributed System Security Symposium (NDSS)*.

[61] Vivek Thampi. 2014. Udis86 Disassembler Library for x86 / x86-64. http://udis86.sourceforge.net.

[62] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G. Bringas. 2015. SoK: Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*. 659–673.

[63] Giovanni Vigna. 2017. When Malware is Packing Heat. LastLine. https://www.lastline.com/labsblog/malware-packing.

[64] Christina Voskoglou, Jed Stephens, Konstantinos Korakitis, Michael Condon, Richard Muir, and Simon Jones. 2021. *State of the Developer Nation: 21st Edition*. Technical Report. SlashData.

[65] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-based Fault Isolation. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*. 203–216.

[66] Fish Wang and Yan Shoshitaishvili. 2017. Angr – The Next Generation of Binary Analysis. In *Proceedings of the IEEE Secure Development Conference (SecDev)*. 8–9.

[67] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making Reassembly Great Again. In *Proceedings of the 24th Annual Network & Distributed System Security Symposium (NDSS)*.

[68] Shuai Wang, Pei Wang, and Dinghao Wu. 2015. Reassembleable Disassembling. In *Proceedings of the 24th USENIX Security Symposium*. 627–642.

[69] Shuai Wang, Pei Wang, and Dinghao Wu. 2016. UROBOROS: Instrumenting Stripped Binaries with Static Reassembling. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 236–247.

[70] Wenhao Wang, Xiaoyang Xu, and Kevin W. Hamlen. 2017. Object Flow Integrity. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. 1909–1924.

[71] Richard Wartell, Vishwath Mohan, Kevin Hamlen, and Zhiqiang Lin. 2012. Securing Untrusted Code Via Compiler-agnostic Binary Rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*. 299–308.

[72] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*. 157–168.

[73] Xiaoyang Xu, Masoud Ghaffarinia, Wenhao Wang, Kevin W Hamlen, and Zhiqiang Lin. 2019. ConFIRM: Evaluating Compatibility and Relevance of Control-flow Integrity Protections for Modern Software. In *Proceedings of the 28th USENIX Security Symposium*. 1805–1821.

[74] Carter Yagemann, Matthew Pruett, Simon P. Chung, Kennon Bittick, Brendan Saltaformaggio, and Wenke Lee. 2021. ARCUS: Symbolic Root Cause Analysis of Exploits in Production Systems. In *Proceedings of the 30th USENIX Security Symposium*. 1989–2006.

[75] Chao Zhang, Mehrdad Niknami, Kevin Zhijie Chen, Chengyu Song, Zhaofeng Chen, and Dawn Song. 2015. JITScope: Protecting Web Users From Control-flow Hijacking Attacks. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*. 567–575.

[76] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical Control Flow Integrity and Randomization for Binary Executables. In *Proceedings of the 34th IEEE Symposium on Security & Privacy (S&P)*. 559–573.

[77] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Security Symposium*. 337–352.