Janitizer: Rethinking Binary Tools for Practical and Comprehensive Security

Mahwish Arif
University of Cambridge
Cambridge, United Kingdom
mahwish.arif@cl.cam.ac.uk

Sam Ainsworth
University of Edinburgh
Edinburgh, United Kingdom
sam.ainsworth@ed.ac.uk

Timothy M. Jones
University of Cambridge
Cambridge, United Kingdom
timothy.jones@cl.cam.ac.uk

Abstract

Comprehensive application security can only be ensured if all code that it is going to execute is protected: any unprotected code, either from libraries or the application, becomes a potential attack surface. Compilers contain extensive suites of tools to aid in this, but require source availability that is often infeasible. Existing static and dynamic binary rewriting techniques that retrofit for security either lack in code coverage or soundness, or incur very high performance overhead.

We present a case for adopting hybrid static-dynamic mechanisms to ensure comprehensive security for binaries, providing sound and practical solutions. We highlight the limitations of existing hybrid tools in their use for security purposes, and provide insights to re-architect them. We provide a framework, Janitizer, that enables sound and comprehensive code coverage for entire applications, presenting hybrid binary implementations for two important classes of security schemes; a memory sanitizer and a control flow integrity scheme. These achieve the coverage and correctness of high-overhead dynamic techniques, while maintaining performance levels of low-coverage static techniques.

CCS Concepts: • **Security and privacy** → *Software reverse engineering*; *Software and application security*.

Keywords: Binary analysis, Binary security, Static/Dynamic analysis

ACM Reference Format:
Mahwish Arif, Sam Ainsworth, and Timothy M. Jones. 2025. Janitizer: Rethinking Binary Tools for Practical and Comprehensive Security. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25), March 01–05, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3696443.3708930

1 Introduction

Defending against software security vulnerabilities remains a prevalent problem despite decades of research effort [2, 4, 6, 13, 34, 37, 47, 49, 57]. Large codebases exist in legacy and system-level software (e.g. compilers, runtimes, browsers) in low-level, unsafe languages, such as C/C++, which lack type safety, bounds checking or garbage collection [52]. This leaves them vulnerable to exploits (e.g. buffer overflows [21]) that perform memory corruption or control-flow hijacking.

Even worse, these vulnerabilities may be in software the user relies on but does not have source-code access to. For example, they may rely on third-party applications and shared-object libraries [32] that are untrustworthy due to the programming language they are written in, and unverifiable due to their closed source. In this case, we can neither enforce safe languages and programming practices during development, nor perform compile-time vulnerability analysis and hardening. Our only option is to *retrofit* the binaries themselves with security policies. This may also be the only practical option with large or legacy software, where recompilation is either too time consuming or relies on obsolete tools.

Several existing techniques aim to retrofit binaries with various security protections. These use underlying binary analysis and rewriting frameworks [5, 9, 10, 14, 26, 35, 36, 39, 40, 50, 51, 54, 55, 60, 61] to either *statically* analyze and rewrite the binary to insert security monitoring code [18, 19, 23, 41] or do so *dynamically* [15, 48, 59] by observing code at run-time. Static techniques struggle with precise control-flow recovery, and are consequently unsound or limited to a subset of binary classes [23], whereas dynamic tools incur prohibitively high performance overhead [48], with limited scope for cross-block optimizations.

To obtain strong guarantees from real binaries, we need all-encompassing, correct code coverage at the same time as high enough performance for real-world use. We argue that static-dynamic hybrid mechanisms are the only way forward. For example, we can combine insights from complex static analysis for optimization opportunities with run-time code disambiguation for soundness. Earlier works [60] feature hybrid mechanisms that can combine information in this way. However, their focus on hot-code optimizations leaves them unable to handle analysis that covers entire programs.

We re-architect hybrid frameworks to allow enforcement of *comprehensive binary-security policies*, by identifying different categories of code we need to support, showing how analysis and optimization we can do at static- versus dynamic-time varies, and build an engine to disambiguate the various

CGO '25, March 01–05, 2025, Las Vegas, NV, USA

ACM ISBN 979-8-4007-1275-3/25/03
https://doi.org/10.1145/3696443.3708930

cases. We then show how our framework, Janitizer, supports real security policies. We make the following contributions:

- We identify shortcomings in existing static, dynamic *and* hybrid binary tools and solutions, in achieving practical and comprehensive security for binaries.
- We propose a static-dynamic hybrid approach to binary security that ensures soundness and coverage for all binary code, including statically generated position-dependent and independent code, or code dynamically linked, loaded or generated. To this end, we develop Janitizer, a binary tool that employs tailored strategies for different types of code, depending on their suitability for static analysis.
- We provide hybrid binary implementations of a memory sanitizer JASan, and a control flow integrity scheme JCFI, in Janitizer to show how it can be used, and the challenges a hybrid mechanism can overcome.

For JASan, We show 3× performance overhead, almost identical to existing static tools [23] and yet with full coverage only previously possible with dynamic-only techniques [40] that incur 9.8× overhead. For JCFI, we show comparable performance to existing static binary techniques (1.29× overhead) while providing over 99.7% reduction in control-flow-integrity targets, compared to 93.3% and 98.8% for existing schemes Lockdown [44] and BinCFI [59].

2 Motivation

In order to detect, prevent or mitigate a security attack on an application, we need to protect all the code a user application executes. Anything unprotected, such as a shared library, or dynamically generated or loaded code, becomes a potential attack surface. Existing techniques that analyze and transform binaries for security either lack coverage, incur very high overhead, or are unsound, owing to the limitations of the analysis and rewriting modes they employ.

2.1 Static: Low-Coverage or Unsound

Static binary frameworks facilitate analysis [17, 24, 28] and rewriting [5, 9, 23, 26, 35, 39, 42, 54] of a binary without the need to execute it first. This avoids overhead at run-time, allowing complex cross-basic-block analysis and optimization.

In order to generate binaries that are both sound (do not break working code) and complete (do not miss out parts of the application, or fail to modify certain classes of code), static frameworks need to perform not only accurate disassembly and recovery of the application's control-flow graph (CFG), but also accurate reconstruction of control flow for the rewritten or instrumented binary. This is a largely unresolved issue both due to the nature of binary code [38] and limitations inherent to static techniques, such as 1) absence of semantic/higher-level information in binaries, 2) the presence of indirect control-transfer instructions (CTIs) whose targets are not known statically, and 3) undecidable differentiation between constants and references due to inter-mixed data and code [30]. The result is that static techniques either cannot guarantee both soundness and completeness, or are limited in their scope to only small classes of binaries compiled in very specific ways [23]. Techniques [45, 46] that claim to work despite these limitations either rely on conventions often not followed by hand-written or low-level library code, such as specific caller/callee-saved code patterns, or simply fail gracefully when their heuristics do not hold.

Retrowrite [23], a framework used to implement AddressSanitizer [47] for binaries, uses *reassembleable assembly* and *symbolization* [56] to address the issue of indirect calls and to distinguish constants from references. However, in order to avoid heuristics and produce a correct binary in all cases, it relies on relocation information being available to *symbolize* targets, so is limited to binaries compiled as *position-independent code* (PIC), and is inapplicable to C++ code containing exception handling. BinCFI [59], a static control flow integrity (CFI) scheme, similarly uses symbolization, without being limited to PIC code, but suffers from unsoundness due to code-data disambiguation being undecidable in theory—without dynamic analysis—and impractical in practice [9, 43], giving incorrect disassembly at full-application scale [23].

Finally, static frameworks are inevitably unable to analyze dynamically generated code, further limiting their coverage.

2.2 Dynamic: Slow and Limited Analysis

Dynamic binary frameworks [10, 14, 36, 40, 55] analyze, instrument or transform a binary during execution. This avoids the limitations of static frameworks, as control flow can be accurately recovered since targets of indirect control transfers are materialized at run-time. Since these tools operate on code during execution, separation of code and data is simple, as is handling dynamically loaded/generated code.

Still, due to analysis and transformation taking place at run-time, dynamic-only tools incur high performance overhead. This makes them impractical for large software as slowdown can become prohibitively expensive. The high overhead also limits them to relatively simple analyses and transformations [40]. This is because dynamic tools discover and/or present one basic block—the smallest control-flow unit—at a time to the transformation engine to make code discovery and control-flow construction simpler, avoiding the pitfalls of static rewriting at the cost of performance [7, 58].

2.3 Existing Hybrid Tools: Not Fit for Purpose

The concept of using a static-dynamic hybrid mechanism for binary analysis or transformation is not entirely new. Several binary frameworks incorporate both static and dynamic mechanisms for analyzing or transforming a binary. In theory, this allows sound run-time control-transfer analysis coupled with complex static analysis and transformation.

However, many frameworks either restrict usage to a single mechanism (static or dynamic analysis) at a time [10], or their hybrid component relies on symbolic execution [50, 51], which is designed for finding bugs in the software during development or testing phase rather than for instrumenting or modifying a binary to observe run-time behavior.

Another framework, Janus [60], features a hybrid approach for binary analysis and modification through statically guided dynamic translation. Its static analyzer generates and encodes hints for the dynamic translator to guide the run-time instrumentation or modification. Although it provides a suitable foundation for a hybrid security analysis mechanism, Janus' focus on hot-code optimization leaves out many important features that are necessary for whole-program-scale analysis and transformation (section 3.2).

3 Janitizer

We develop Janitizer, a framework for comprehensive security analysis of entire applications, that showcases the strengths of employing a hybrid mechanism for security, and addresses the short-comings of existing hybrid binary frameworks laid out in section 2.3. Our goal is to ensure robust yet practical security for diverse classes of binary code that an application can execute, including static code (whether compiled as PIC or non-PIC) and dynamic code (dynamically loaded/generated).

We propose using static analysis to handle complex cross-block analysis and identify instrumentation and optimization opportunities, but defer transformations to run-time, when control-flow ambiguities are resolved and code-data separation is clear. This ensures the soundness of the binary code that is executed, and avoids the pitfalls of static-only techniques—such as restrictions to certain classes of binaries [23] or producing unsound binaries when disassembly or rewriting heuristics do not hold [45, 58]. By offloading the analysis to static time and performing it only once, rather than each time an application runs, its run-time overhead is also removed from application execution time. For code that is either dynamically generated or statically undecided/unseen, we opt for simpler and lightweight run-time analysis.

3.1 The Need for a New Approach

We identify some key requirements needed in our hybrid framework to achieve comprehensive practical security. First, the framework should support detailed static analysis for *all* statically available code, whether compiled as position-independent or position-dependent, and feed the analysis results to a dynamic tool for run-time transformation. Second, for performant security analysis, the framework should identify code sections (or memory locations) that do *not* require processing by the dynamic modifier. For instance, memory accesses statically proven to be safe need not be instrumented or rewritten, to reduce the overhead. To support this, the dynamic modifier should be able to differentiate between the code that has been statically seen and analyzed to code that appears only at run-time. Third, the dynamic modifier should leverage the results from the static analyzer when transforming the binary. In addition, any code left out of the stronger static analysis stage, either because it is dynamically loaded (e.g. via `dlopen`), dynamically generated, or simply not discovered at the static analysis stage, should undergo a simpler dynamic analysis before modification. We must ensure that all code is analyzed and modified, prioritizing static analysis but falling back to run-time analysis when necessary. To this end, we add hybrid approaches *on top of* existing hybridized execution environments: using a static component with strong analyses for code we can analyze offline, a dynamic modifier to efficiently and correctly transform code even in the presence of complex control flow, combined with simpler run-time analyses for code only discovered or generated dynamically.

3.2 Building on Janus

To this end, we reuse some of the ideas employed in Janus [60], which, in theory, is a good starting point for whole-program dynamic security analyzers. However, there are several limitations in Janus that inhibit the use of its hybrid capabilities to support security coverage of real analysis techniques, such as AddressSanitizer [47], for the diverse types of code that make up full applications. Janus, designed with a focus on performance optimization through parallelization of hot loops, leaves out many code sections and blocks from its detailed static analysis, where it considers them irrelevant or uninteresting, and ignores all other code as needing no treatment. This is inadequate for security applications, as comprehensive coverage of *all* code is imperative in such contexts, and yet we need to distinguish statically proven safe code, statically instrumentable code, and unseen code. Another limitation lies in Janus' *rewrite schedule*, the primary interface that allows use of static analysis results by the dynamic modifier. It assumes the binary to be compiled as position-dependent (i.e. non-PIC), thereby lacking the support for utilizing results of static analyses for position-independent code (typically used for libraries, and more recently for the main binary executable as well). Given the requirements earlier, we carry out an architectural overhaul to overcome all these limitations to develop Janitizer.

Figure 1 shows the workflow of Janitizer, featuring its static analyzer and dynamic modifier that support diverse analyses and transformations over diverse code types. In the following sections, we describe in detail the static and dynamic components of Janitizer that handle different classes of binaries to ensure comprehensive security coverage, while keeping the performance overhead low.

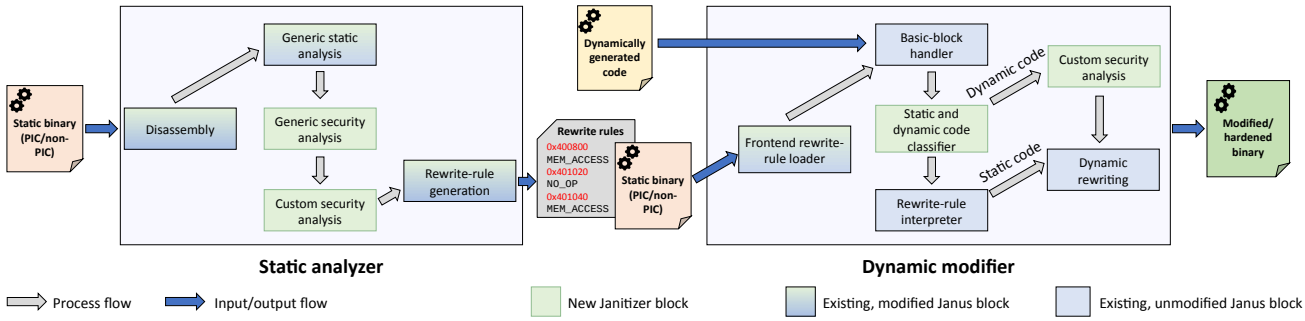
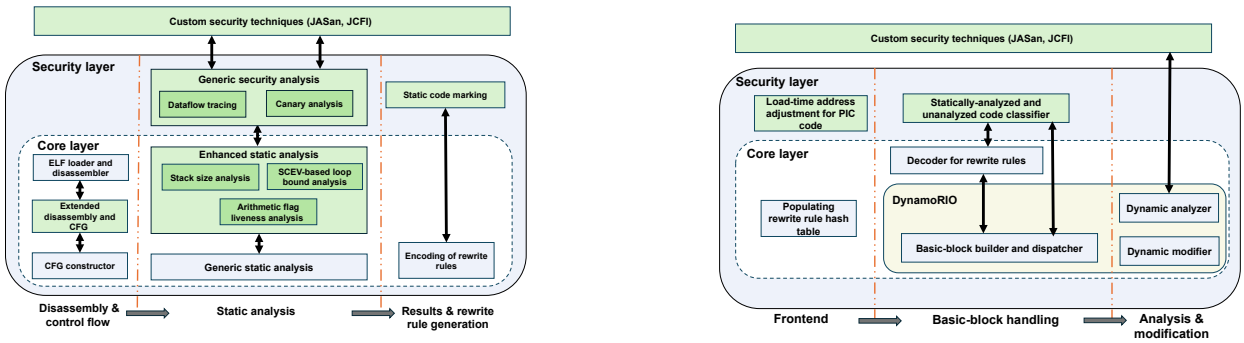


Figure 1. Workflow of Janitizer: the static analyzer takes statically generated PIC or non-PIC binary code, disassembles it, performs generic as well as custom (user-defined) security analyses for a target security technique, and generates rewrite rules for a dynamic modifier. Next, the front-end of the dynamic modifier decodes these rules and builds correspondence between static and load-time addresses to handle analysis results for both PIC & non-PIC code. All code to be executed is input here, with a classifier differentiating between previously statically seen and newly dynamically observed/generated code. For statically analyzed code, transformation or instrumentation is performed based on the rewrite rules, whereas for dynamically observed code, custom dynamic analysis (based on a second set of user-defined analyses) is performed before it is modified.



(a) Static analyzer **(b)** Dynamic modifier

Figure 2. Janitizer's static analyzer (a) and dynamic modifier (b). Green blocks show modules added to Janus [60] to enable comprehensive binary security. Double-sided arrows indicate functionally related blocks. Horizontal arrows represent workflow.

3.3 Stronger Analyses for Static Binaries

Figure 2(a) illustrates the design of Janitizer's static analyzer, featuring layers that support core and security-specific functionalities. The core layer reuses components from Janus where feasible, but with enhanced disassembly and control-flow construction for the whole binary, and provides new helper analyses to improve correctness and performance of whole-program analysis. These a) aid disambiguation between statically seen and statically unseen code in the dynamic modifier and b) implement and facilitate various types of security analysis. Any custom security techniques, such as AddressSanitizer [47] and Control-Flow Integrity [2] featured in this paper, can then use the functionality provided by the core and security layers. Here we describe the key features of Janitizer's static analyzer.

3.3.1 Control-flow recovery/analysis for all code. The binary code belonging to each statically available module from an application is individually presented to the static analyzer, where it first undergoes disassembly, similar to Janus [60]. Whereas Janus skips building basic blocks and other control flow structures for code sections other than `.text`, as it deems them uninteresting for parallelization, we instead extend control-flow construction to *all* executable code sections, including the procedure linkage table (PLT), initialization (`.init`) and finalization (`.fini`) sections. This is to ensure that these are available for subsequent analysis stages that require cross-block analysis, so that for security we analyze all code that an application may execute, and for performance, as much as possible should be at static time.

After disassembly and control-flow recovery, the code undergoes further static analysis, including new generic security and enhanced code analyses. These analysis stages cover all executable code blocks, unlike in Janus, which excludes functions without loops from further analysis and does not consider code blocks unreachable from a function's entry node (e.g. being the target of indirect control flow). All

Rule ID	BB Addr	Instr Addr	Optional Data			
			Data1	Data2	Data3	Data4

Figure 3. Format of rewrite rules.

of the shared-object-library dependencies are found using the *ldd* Unix tool, and also undergo the same analysis.

Lastly, any transformations identified for a particular instruction or control-flow structure by the security analysis need to be passed on to the dynamic modifier. These are encoded as rewrite rules (format shown in figure 3) for a corresponding instruction and the enclosing basic block, and are recorded in separate files for each binary module presented to the static analyzer and loaded at run-time with the module. Each rewrite rule corresponds to a handler routine in the dynamic modifier that performs transformation. This means that static analysis for a shared object library dynamically linked by multiple binaries needs to be performed only once. Figure 3 shows the format of a rewrite rule.

3.3.2 Enhanced static analysis. Janitizer adds several new analysis routines to the framework that can unlock opportunities for rewrite-rule optimization, benefiting not only security-related schemes but also enhancing the overall performance of generic binary applications. For example, the modification or instrumentation of binaries can affect the status of arithmetic flags (as we show in JASan, section 4.1), necessitating preservation. For performance, we employ liveness analysis to identify code locations where these flags are not live at the point of instrumentation, to avoid saving and restoring them. In cases where exact control flow cannot be determined statically, such as indirect branches, we assume that all arithmetic flags are live so need preservation. Janitizer also leverages scalar evolution analysis (SCEV) [3] to examine loop bounds. This aids in statically identifying memory accesses that are either loop invariant, requiring one check at the start of the loop, or linked to the loop iterator so cannot go out of bounds given loop-bound information.

3.3.3 Support for generic security analysis. We add generic analyses that can be used as building blocks for security mechanisms. Canary analysis works out which code represents stack canaries [8], and thus must not be disturbed by code modification, and may be reused to facilitate further program analysis such as poisoning (section 4.1) or randomization of canary location [27]. SSA-level diffuse-chain tracing, traces the chain of dependencies between nodes, to determine, for example, whether an instruction accesses a memory location that was allocated at a particular dynamic memory allocation site (`malloc`), or to monitor the flow of untrusted data as seen in taint-tracking mechanisms.

3.3.4 Marking statically inspected code. For each basic block seen by our static analyzer, there are two possibilities when presented to the dynamic modifier: 1) it needs transformation or instrumentation based on the rewrite rules, or 2) it remains unmodified. In a scenario where comprehensive coverage of all code (both static and dynamic) is not required, as is the case for Janus [60], a basic block with no transformation can simply be left without a rewrite rule. This minimizes the number of rewrite rules when the common scenario is that a block remains unmodified or unoptimized, and allows the dynamic modifier to ignore it. This contrasts with Janitizer, where the common case is that a block *is* instrumented or modified for security. For Janus, any new basic block that appears in the dynamic modifier without an associated rewrite rule will be perceived exempt from further analysis, and translated as-is. In Janitizer, a block without a rewrite rule may instead be a new dynamically discovered block that needs instrumentation.[1] To address this, we introduce a mechanism, called *no-op* rules, that marks statically inspected code to assist the dynamic modifier to disambiguate between blocks that need no further analysis (statically proven safe) and blocks not yet analyzed (because they were never encountered statically). These rules get used by the dynamic modifier at run-time (figure 4).

3.4 Run-time Modification, and Analysis Coverage for Dynamically Available Code

To guarantee the soundness of instrumented binaries regardless of control-flow and code-data ambiguity [23, 60], Janitizer adopts run-time modification even for statically seen code. We re-use the idea of statically guided dynamic modification from Janus [60], albeit with several new mechanisms to ensure that the dynamic modifier in Janitizer can 1) utilize rewrite rules generated by the static analyzer for the main executable and all its dependencies whether compiled as position-independent (PIC) or non-PIC code [23] (unlike Janus, which supports non-PIC only), 2) distinguish statically seen code from statically unseen code, and 3) employ fallback analysis for code only seen by the dynamic modifier.

Figure 2(b) shows the layered structure of Janitizer's dynamic modifier that incorporates these mechanisms for comprehensive security coverage. The core layer features functionalities for loading and decoding the statically generated rewrite rules for each module, building basic blocks for code presented for execution, and performing modification and transformation of these blocks according to the rules. The security layer adds new mechanisms to identify and handle

[1] There are multiple scenarios where a basic block may not have been seen by the static analyzer: 1) dynamically generated code, 2) shared-library dependencies that cannot be determined statically using tools like *ldd* (accounting for up to 40% of shared-library code [58] in real applications) because they are loaded through `dlopen`, and 3) basic blocks not discovered by the static analyzer due to incomplete control-flow recovery. If a shared object library is loaded during execution via `dlopen` and happens to have an associated file with rewrite rules, they can be processed by the corresponding handler routine, otherwise we fall back to dynamic analysis.

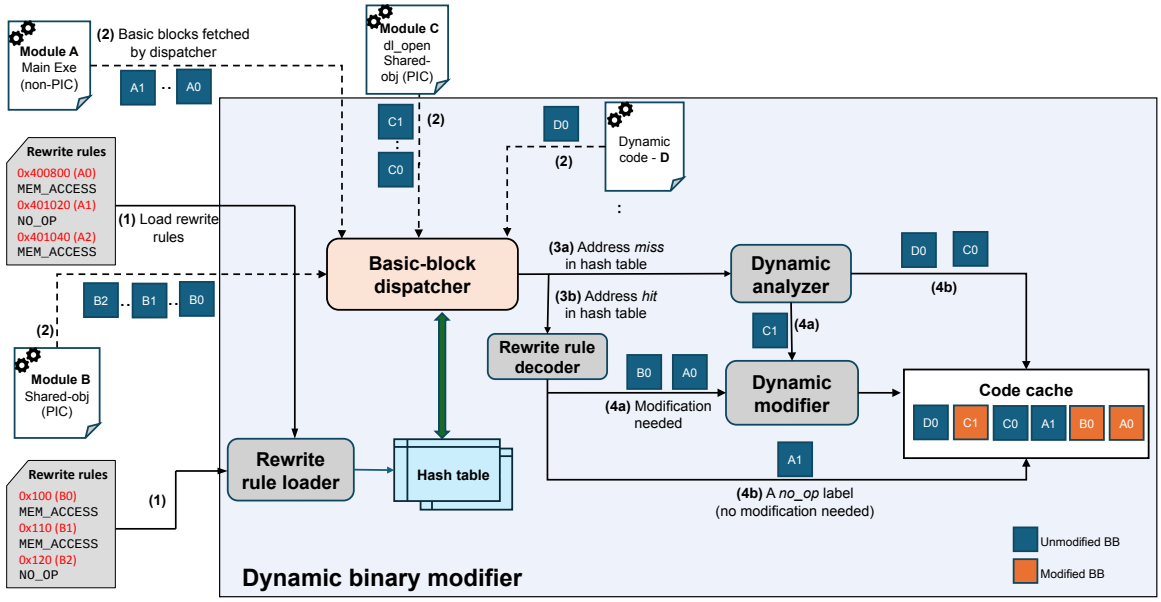
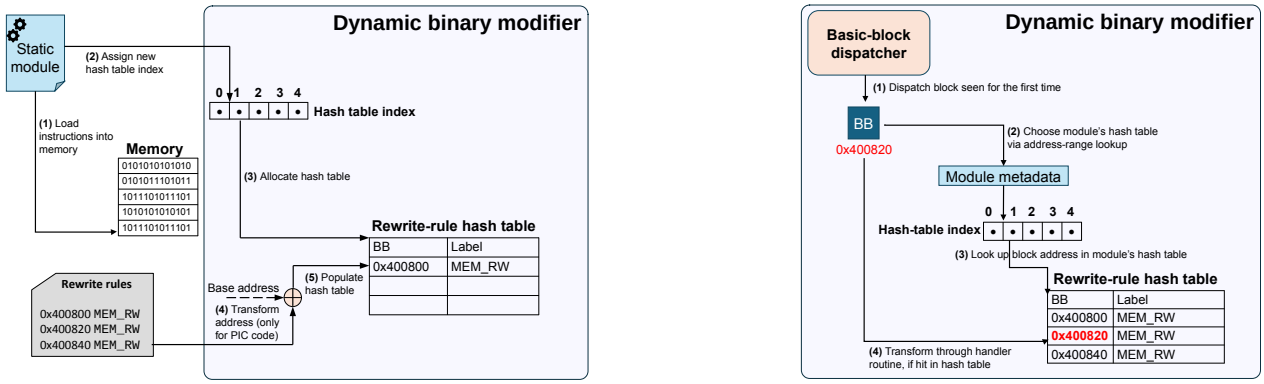


Figure 4. Janitizer's dynamic modifier decides how a particular basic block (BB) needs to be treated. The rewrite rules for statically analyzed modules (A and B) are loaded into hash tables (1). The BB dispatcher builds the BBs (2), and checks if they have associated rewrite rules in the relevant module's hash table. If a BB address *hits* in the hash table (3b), it is forwarded to its respective handler routines into dynamic modifier based on its label. If the address is a miss (e.g. in case of `dl_open`ed module C or dynamic code D), it is forwarded to the dynamic analyzer, which checks whether it needs further modification.



(a) Binary-module load time **(b)** Basic-block dispatch time

Figure 5. Janitizer's support for (a) loading and (b) using statically generated rewrite rules from multiple modules at a time, including main executable and shared-object dependencies whether compiled as position-independent (PIC) or non-PIC code.

the diverse types of code that it must support. We further introduce plug-ins atop these layers to implement custom security techniques (mirroring the static component of the framework figure 2(a)). This is to allow additional (typically simpler) analysis for code only seen dynamically, unlike in Janus, which only optimizes code seen by the static analyzer.

3.4.1 Overall dynamic binary modifier.

Figure 4 gives an overview of the whole dynamic binary modifier. When a new statically analyzed module is loaded (PIC or non-PIC), its rewrite rules are read and written into the module's hash table (1), as described in section 3.4.2. When a basic block from a module is first seen (i.e. it is targeted by a control transfer instruction), the dispatcher fetches it (2) and consults the relevant hash table. This occurs for all types of code—the main executable (module A), a shared object library that's been statically analyzed (module B), a shared object library loaded through `dl_open` (module C) or dynamically generated code (D). On a miss in the hash table (3a), indicating no rewrite rules present, the basic block undergoes dynamic analysis. Note that the exact nature of the dynamic analysis to be

performed depends on the custom security technique being implemented, as described later in section 4.1.1 for JASan and section 4.2.2 for JCFI. On a hit (3b), the rewrite rules are decoded. These rewrite rules (or the dynamic analyzer) may determine that modifications are required (4a), in which case the dynamic modifier makes the appropriate changes before placing the basic block in the code cache for execution. On the other hand, if the rewrite rule is a no-op rule, or the dynamic analyzer decides no changes are necessary (4b), the basic block can be placed in the code cache as-is.

3.4.2 Support for multiple PIC/non-PIC modules. Figure 5(a) shows how we handle the rewrite rules produced by the static analyzer for multiple modules of different types. Janitizer stores hash tables per-module and adjusts the precomputed addresses of the rewrite rules based on where modules are loaded into the memory. When a new module is encountered, (1) instructions are loaded into memory and the dynamic modifier obtains a new hash table index (2) from its map of modules, and a new hash table is created (3). Rewrite rules are then loaded into this hash table, and decoded for further processing by corresponding handler routines (4). As these addresses were determined statically, they do not reflect the runtime address of an instruction or basic block for position-independent code. Therefore, for PIC binaries, we adjust the instruction or basic block address according to the memory address it is loaded at, before adding the rewrite rule to the module's hash table[2].

Later, when a basic block is first seen by the binary modifier, the relevant hash table is consulted to determine the rewrite rules necessary to apply, shown in figure 5(b). When a newly seen basic block is dispatched for the first time (1), its corresponding module is identified (2) so that the block's address can be looked-up in the correct hash table (3) to determine its rewrite rule to apply appropriate modifications(4). Alternatively, if there are no rewrite rules for this basic block, then it is taken forward for dynamic analysis.

3.4.3 Custom security analysis for dynamic code. In order to provide comprehensive protection for an application, we also support dynamically generated code. JavaScript code (just-in-time compiled within web browsers often written in C/C++) is a common use case. As dynamic code is not available or seen with static binaries, it can only be analyzed, modified and instrumented dynamically. Dynamically loaded libraries with no rewrite rules attached, and undiscovered static code, are covered identically (section 3.3).

Since the dynamic modifier presents code one basic block at a time (because this is when the code is first discovered to be executed and thus unambiguously not data), in contrast to the global overview available to static analyzer, the analysis is likely to be implemented not only differently from the equivalent static technique, but also with simpler analysis to avoid the high run-time overhead of complex analysis. For example, our dynamic JASan analyses all loads and stores it sees, whereas the static version can find accesses that are statically safe through between-block analysis (section 4.1). This means custom security techniques need to provide two different plug-in passes: one for the static analyzer, able to do cross-block analysis, and one for the dynamic analyzer, which can only work at the basic-block level, as a fallback.

[2]Since addresses in the hash tables are adjusted at load-time, even though modules may reuse addresses, there will be no overlap between different hash tables at run time: any run-time address will exist in at most one hash table. Still, we keep them in separate hash tables to allow modules to be loaded and unloaded efficiently without scanning to remove stale hints, even if different modules are loaded at the same address at different times.

4 Examples

To demonstrate the flexibility of Janitizer, we develop binary implementations of two complementary security techniques within the framework: an address sanitizer for memory safety and a control-flow-integrity scheme.

4.1 JASan: Binary Address Sanitizer

AddressSanitizer [47] is one of the most popular memory-safety techniques and has been integrated into the LLVM compiler as an instrumentation pass combined with a runtime library. It detects violations by placing poisoned *redzones* around buffer regions that should not be addressed, with a shadow memory to keep the allocation status (poisoned/unpoisoned) of each byte of a process' memory. Each load and store is instrumented with a run-time check of the shadow, and the memory allocation/de-allocation calls are diverted to a special memory allocator from LLVM ASan's runtime library using `LD_PRELOAD` to poison heap objects.

4.1.1 Overview. Our version, JASan, is inspired by the sanitizer in Retrowrite [23], in that it provides full heap-object protection, enforces stack protections at the coarse granularity of a stack frame by identifying canaries (section 3.3), and ignores global regions due to a lack of higher-level type information. JASan works with both PIC and non-PIC code, unlike Retrowrite (section 2.1). Owing to its dynamic-only fallback, JASan can also provide security guarantees for dynamically loaded/generated code.

JASan splits work between dynamic and static passes. The static analyzer, after disassembly and control flow construction, identifies code locations with memory addresses that need to be monitored, determines stack canary locations to be poisoned/unpoisoned to protect against stack frame overflows (figure 6 shows an example code and rewrite rule), and precomputes cross-block register- and arithmetic-flags liveness information to reduce the overhead of instrumentation in the dynamic analysis phase. The dynamic pass then performs control-flow disambiguation and instruments statically analyzed code with selective saving and restoring of registers and flags based on pre-computed liveness information. For statically unanalyzed code, it conducts a simpler, per

basic-block analysis, primarily to instrument every memory load or store operation and handle poisoning/unpoisoning of canaries. In this case, it conservatively saves and restores both the arithmetic flags and any registers used by the instrumentation routines, without considering their liveness status (which it does not compute).

We inline instrumentation routines using hand-written meta (non-application) assembly instructions to avoid the overhead of DynamoRIO's[3] clean-calls. For statically seen code, this also allows us fine control over the registers and flags used, letting us utilize the precomputed liveness to save and restore fewer registers around the instrumentation.

4.1.2 Limitation of intra-procedual liveness analysis in special cases. Intra-procedural liveness analysis faces limitations at the procedure boundaries where standard caller-/callee-saved register conventions are not followed, either due to compiler optimizations or hand-written assembly code. We observed one such example in binaries compiled with `gcc -O2` flags (such as `ipa-ra`[4]) where caller-saved registers are not saved/restored if the callee is in the same compilation unit and does not use those registers. In such cases, the intra-procedural liveness analysis in the callee incorrectly concludes that these caller-saved register are free to use (i.e. not live), and leads to the instrumentation routines not saving or restoring them. However, this can lead to issues where the caller later uses these clobbered registers.

Another such scenario is observed in some low-level libraries with hand-written assembly where callee-saved registers are not restored by the callee, and the modified values are later used by the caller. For such cases, we use extended inter-procedural analysis for optimizing, such as saving/restoring those registers just once at the entry and exit of the callee (instead of at every instrumented instruction).

4.2 JCFI: Control Flow Integrity for Binaries

Control Flow Integrity (CFI) [2, 25, 44, 53, 59] is a security mechanism designed to protect against control-flow hijack attacks [11, 33], by validating the targets of control-transfer instructions against a pre-determined control-flow graph.

Forward edges [2] in JCFI are protected using hash-table lookups. We restrict indirect calls to valid function boundaries. Intra-module calls are restricted to the functions defined in the same module, whereas inter-module calls (through the PLT or callbacks) are restricted to the symbols imported by the caller's module and/or exported by the callee's module[5]. Indirect jumps can target within the same function,

```
1  0x40275f:    push  %rbx
2  0x402760:    sub   $0x10,%rsp
3  0x402764:    mov   %fs:0x28,%rax
4  0x40276d:    mov   %rax,0x8(%rsp)
5  0x402772:    xor   %eax,%eax
6  0x402774:    test  %rdi,%rdi
7  0x402777:    je    0x402790
```

(a) Lines 3 and 4 store the canary value from fs:0x28 onto the stack at 0x8(%rsp). The static analyzer identifies the need to poison the canary at the instruction following line 4, i.e. at 0x402772.

RuleID	BB Addr	Instr Addr	Data1	Data2	Data3	Data4
POISON_CANARY	0x40275f	0x402772	0x0	0x0	-	-

(b) The static analyzer encodes information about the dynamic modifier's handler routine, using RuleID, and the address of the basic block and instruction for canary poisoning instrumentation in the rewrite rule.

Figure 6. x86 Assembly code of a basic block with instructions storing canary values on the stack and associated rewrite rule to guide the dynamic modifier to add instrumentation for canary poisoning.

based on jump table addresses, or entry addresses of the functions within the same module (catering for tail-call optimization). For returns, we enforce a precise shadow stack [22]: the intended return address of the function call is pushed at call time and verified at return[6].

Our security analysis involves two steps: first, determining a set of valid targets for each set of indirect control-transfer instructions (CTIs), and second, identifying indirect CTIs where checks must be added by the dynamic modifier.

4.2.1 Static analysis. To determine the set of address-taken functions within a module that indirect calls and jumps are allowed to target, we follow a methodology similar to BinCFI [59] (a static CFI technique). We scan the raw binary for code pointers, using a window of 4-bytes that slides forward by one byte at a time. Given the inherent challenge of statically distinguishing between code and data constants, BinCFI heuristically deems a constant as a code pointer if it falls at an instruction boundary within the code sections. For JCFI, we further refine these constants based on whether or not they match the entry address of a function (i.e. align with a function boundary), information available and pre-determined by Janus' static analysis [60][7]. For PIC binaries where we have offsets with respect to the Global-offset Table

[3]Janus is built on DynamoRIO's dynamic binary modification framework.
[4]The `ipa-ra` flag allows the compiler to break the calling convention by using caller-saved registers for allocation if they are not used by any callee.
[5]We take inspiration from a dynamic-only CFI scheme, Lockdown [44], to dynamically update the list of valid targets for inter-module calls, as the modules are loaded and unloaded, to reduce inter-module targets further.

[6]This policy is the same as enforced by Lockdown [44]; the weaker BinCFI [59] allows returns to go to any call-preceded instruction.
[7]If full symbols are present in the binary, Janus uses function symbols and associated addresses to mark the entries of the functions. In the absence of a full symbol table, it makes use of exported symbols to identify function boundaries, in addition to a cross-block analysis in the static analyzer to infer function start addresses based on the targets of direct calls.

(GOT) instead of absolute addresses, we check if offsets point to a valid function and instruction boundary.

We further identify all the indirect calls, jumps and return instructions where forward and backward CFI verification checks need to be inserted. Additionally, all the call instructions—whether direct or indirect—are also marked for instrumentation to push the return address on the shadow-stack for verification at the return instruction.

4.2.2 Dynamic pass. During the dynamic pass, when a module is loaded into memory, we check for associated static-analyzer hints for adding CFI checks and a set of valid targets for indirect CTIs. If available, these target addresses are populated into hash tables for lookup at run-time[8].

For modules without statically computed information, JCFI performs analysis during the module loading process. This involves a comprehensive scan of the raw binary, similar to in the static analyzer, to identify static code pointers. If a complete symbol table is available, we filter code pointer values based on the corresponding function addresses. In the absence of a full symbol table for such binaries, JCFI falls back to a weaker policy for indirect calls and jumps, relying on exported symbols and code-section addresses, similar to the policy followed by Lockdown [44] for stripped binaries. For basic blocks not seen by the static analyzer, a dynamic-analysis fallback identifies indirect CTIs in the blocks and adding respective instrumentation for CFI checks.

4.2.3 Control-flow abnormalities in low-level libraries. While most of the main executable binary codes adhere to the policies described above, there are some exceptions to that, particularly in low-level shared object libraries such as *libc* and *libgfortran*. One such example is callbacks, where a function address is passed to another module through a call argument, for which the address has not been exported. For such cases, we identify address-taken functions through scanning of the raw binary (section 4.2.1), and add them to the set allowed to be targeted by another module, in addition to explicitly exported symbol addresses.

Another case is when a call instruction targets an instruction not at a detected function boundary in libgfortran. Here, we add target addresses to an allow list, similar to Lockdown. For BinCFI, this is not an issue due to its weaker policy where any code pointer identified through scanning of binary that falls at an instruction boundary is allowed.

Additionally, the routine for lazy symbol resolution in loader (*ld.so*) pushes the function pointer on the stack and then uses return instructions to call the external functions invoked by a binary. As there are just a few instances, BinCFI handles this by modifying the loader itself, thereby replacing these return instructions with an indirect jump. Since BinCFI already allows jumps to target any cross-module function (a weaker policy), this modification is in line with its security policy. For JCFI, we treat these as a special case and attach a forward-CFI (indirect call) verification check, instead of a backward-CFI check.

[8]For PIC binaries, these addresses are adjusted by the module-load address before storing them in hash tables, similar to rewrite-rule handling for PIC.

5 Experimental Setup

We evaluate implementations of JASan and JCFI within Janitizer on an Intel Xeon 6230R system, running Ubuntu 20.04.6, with SPEC CPU2006 [29] workloads to compare like-for-like with previous work [23, 44, 59]. We compile these benchmark workloads with gcc 5.4, a compiler version used for comparison techniques such as Retrowrite, with -O2 flags. We run experiments for performance results a minimum of three times, but see little variation between runs, so error bars are not visible in graphs. We use DynamoRIO version 8.0.0 as part of our dynamic modifier in Janitizer.

We compare the performance of JASan with two state-of-the-art binary address sanitizers, a dynamic-only Valgrind [40] and static-only Retrowrite [23], each running on x86-64. While Valgrind numbers were reproduced and comparable to overheads seen in Retrowrite's evaluation of Valgrind, we were not able to reproduce results for Retrowrite itself as the resulting rewritten binaries were broken, even when reproduced on the operating system, libraries and compiler specified in the original paper. Therefore, we report the overheads for Retrowrite as published in the original paper.

For JCFI, we compare its performance and security protections against the static-only BinCFI [59] and the dynamic-only Lockdown [44]. Since both of these comparison techniques were only implemented for 32-bit x86, we run these comparisons on x86-32 binaries and shared libraries[9]. The public implementation of BinCFI is provided as a virtualized environment, and we found the tool gave segfaults outside this, even with exactly the same versions of compilers and libraries. We thus evaluate it inside the virtual machine, with reference to a baseline inside the same virtual machine.

6 Evaluation

We evaluate Janitizer based on its run-time performance, code coverage and security properties when compared against static and dynamic-only schemes for JASan and JCFI.

6.1 JASan

Figure 7 shows the performance of two versions of JASan and the comparison techniques. JASan-dyn is a dynamic-only version of JASan, without any static analysis. JASan-hybrid is a hybrid and optimized version that uses register and flag liveness information from static analysis[10].

[9]We found the null-client overhead for DynamoRIO in 64-bit mode, when it performs no analysis, to be high relative to the literature. Bruening and Zhao [16] claim 8% and 13% for x86-32 and x86-64 on SPEC CPU2006, whereas we saw 9% and 30%. We expect future optimizations to DynamoRIO to better handle modern code and Janitizer will benefit accordingly.

[10]Retrowrite also uses intra-procedure liveness analysis.

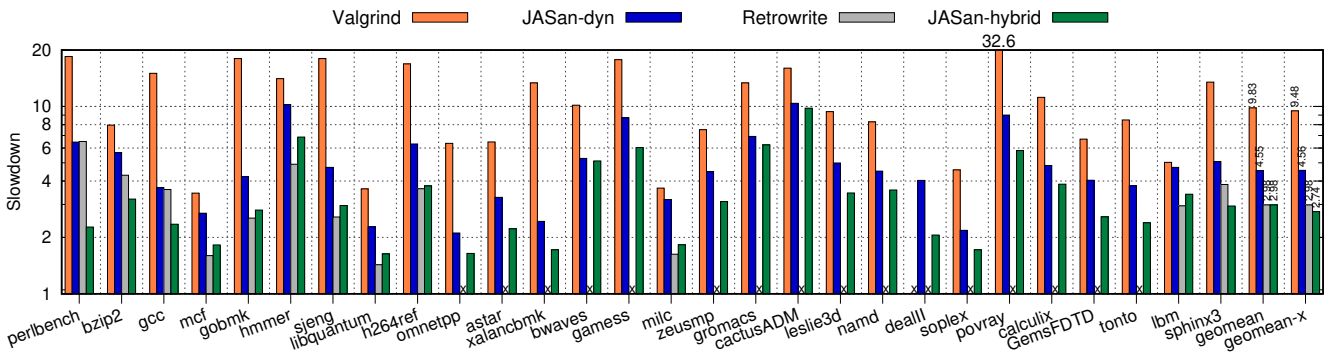


Figure 7. Janitizer overhead of JASan (binary ASAN) on SPEC CPU2006 [29], compared to native execution of benchmarks.

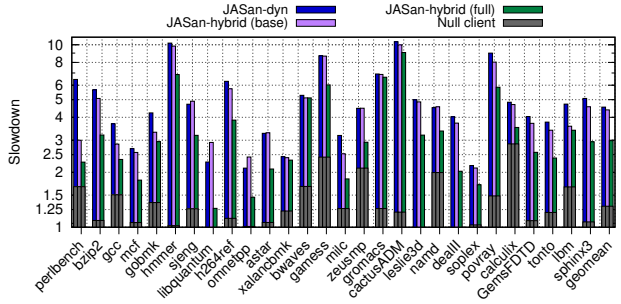


Figure 8. Overhead breakdown of JASan.

6.1.1 Performance. JASan-hybrid incurs 2.98× overhead, relative to native execution, across SPEC2006 benchmarks, compared to 2.98× for Retrowrite and 9.83× for Valgrind[11]. Note that the JASan implementation covers the code of the main executable, which was compiled as non-PIC, as well as of shared libraries, compiled as PIC. As the applications in JASan run under DynamoRIO, we further show JASan's overheads atop DynamoRIO's translation overhead in figure 8 when compared against native execution. The figure also shows JASan-hybrid against a baseline hybrid version that conservatively saves/restores all the registers or flags used by the instrumentation code. Overall, our optimized hybrid implementation sees 27% improvement.

6.1.2 Security properties. We evaluate the security protections of JASan-hybrid in terms of four key metrics: true positives (TP), where it correctly identifies memory safety violations; true negatives (TN), where it correctly recognizes the absence of violations; false positives (FP), where it incorrectly reports violations that are absent and false negatives (FN), where it fails to report actual violations. We evaluate this using the NIST Juliet benchmark suite [12], specifically focusing on vulnerability class CWE122, which includes heap-to-heap, stack-to-heap and heap-to-stack buffer overflows. Since some vulnerabilities may not manifest without specific inputs [12], we only run test cases that require either no input or trigger a vulnerability without any specific input. Figure 10 presents the results separately for good (well-behaving) and bad (with violations) variants of each test case. For bad variants, FN represents the number of test cases where either no or fewer-than-actual violations are reported. For JASan, all FNs represent cases where it only reports heap-to-stack buffer overflow that overwrites stack canary values, consistent with our stack protection policy. For heap overflows, it reports all violations correctly. In contrast, Valgrind fails to detect heap-to-stack overflow for 96 test cases, and reports fewer-than-actual overflows on the heap in 24 test cases. We exclude Retrowrite from this analysis as the generated binaries failed to run correctly.

6.2 JCFI

6.2.1 Performance. Figure 9 shows a performance comparison of JCFI with Lockdown and BinCFI. JCFI incurs a 1.29× slowdown (1.37× without static analysis) compared to 1.21× for dynamic-only Lockdown[13] and 1.22× for static-only BinCFI. Lockdown failed to run on omnetpp and dealII—an issue also reported in the original paper. Similarly, BinCFI-generated binaries did not run for gamess and zeusmp. In order to have a fair comparison with BinCFI, which does not implement a shadow stack, we further breakdown the overhead of JCFI coming from forward CFI and backward CFI (the latter implemented using shadow stack) in figure 11, showing that it incurs a 1.15× overhead when only forward CFI is applied. This makes JCFI's performance similar to BinCFI, slightly lower than Lockdown, but as we show next, delivering considerably stronger security guarantees.

[11] We show geomean for all the benchmarks that successfully run for each scheme, as well as only benchmarks that run for all schemes (`geomean-x`).

[12] The Juliet suite is primarily designed for static analysis tools like vulnerability analyzers or fuzzers that either detect vulnerability patterns or generate inputs to exploit vulnerabilities.

[13] Lockdown's paper reports 1.32× overhead. When contacted, an author suggested a bug may have resulted in missed CFI checks in the open-source version. As Lockdown uses a custom secure loader, we were unable to directly compare the number of checks actually performed by JCFI and Lockdown, beyond the AIR metrics self-reported by the Lockdown tool.

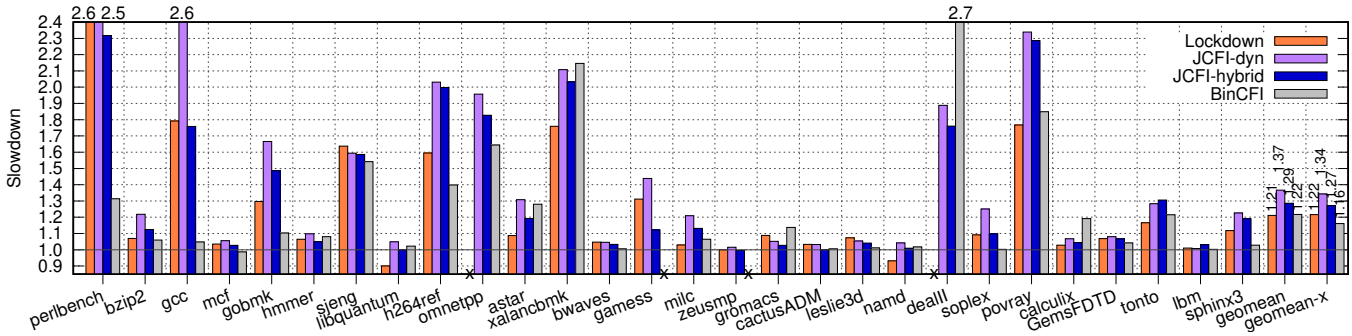


Figure 9. Overhead of JCFI, Lockdown and binCFI on SPEC CPU2006, compared to native execution of benchmarks.

		Valgrind	JASan
good variant	**False Positives**	0	0
	True Negatives	624	624
bad variant	**True Positives**	504	528
	False Negatives	120	96

Figure 10. Security properties across 624 Juliet NIST CWE-122 test cases. Each test case has a *good* variant (well-behaving) and a *bad* variant (with violations).

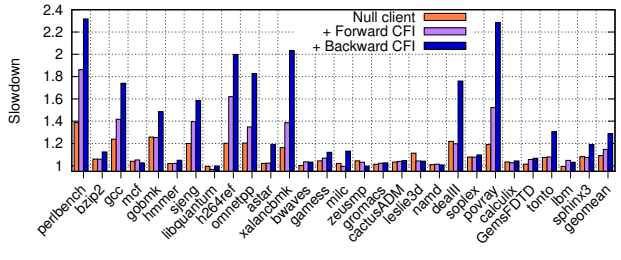


Figure 11. Combined contribution of forward/backward CFI (shadow stack) towards JCFI-hybrid overheads.

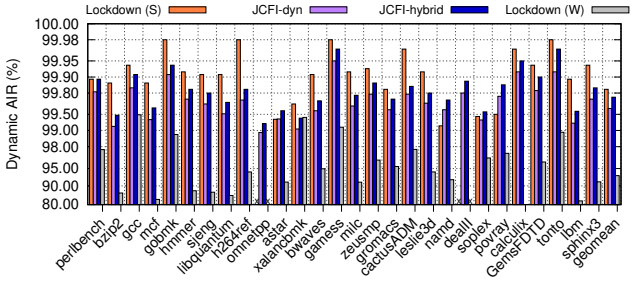


Figure 12. Avg. dynamic indirect-target reduction (DAIR) of JCFI, and Lockdown Strong (S)/Weak (W) (higher better).

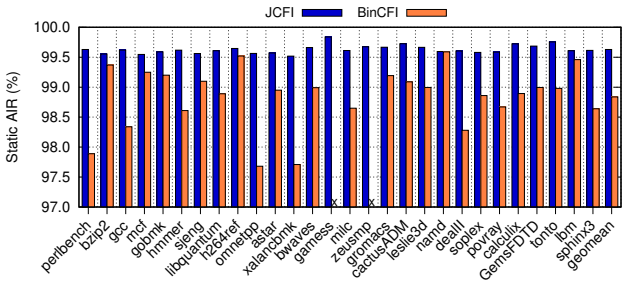


Figure 13. Average static indirect-target reduction (AIR) metric of JCFI-hybrid and BinCFI (higher better).

6.2.2 Security-related properties.

We show that JCFI offers better protection against CFI attacks compared with BinCFI and Lockdown, both soundness and completeness.

Soundness. A sound scheme does not generate false positives i.e. all the bugs or violations it reports are actual violations, and not correct or intended behavior. Lockdown generates false positives for inter-module indirect calls made through callbacks in h264ref, cactusADM and gcc. Its default *strong* policy allows inter-module indirect calls only if the target is both explicitly imported by the source module and exported by the destination modules. For callbacks that do not follow this policy, Lockdown uses heuristics to identify potential valid targets, but these miss many cases. For instance, in h264ref and cactusADM, function pointers of comparison routines are passed to libc's `qsort` function via stack, which is missed by the Lockdown's heuristics. Similarly, it misses a code pointer in the same function due to its sliding 4-byte window with no fallback. In contrast, JCFI allows inter-module transfers to exported symbols as well as any identified address-taken functions in the destination module. JCFI avoids these false positives by using cross-block static analysis analysis to build a control-flow graph and identify function boundaries, something not possible with the dynamic-only approach in Lockdown.

Completeness. The completeness of a security mechanism is determined by its effectiveness to reduces the number of false negatives i.e. the potential violations of the security protection. For this purpose, we use Average Indirect Target Reduction (AIR), a metric commonly used in the literature [44, 59] to gauge the percentage of targets removed from the list of potential targets for a ROP attack[14].

[14]For example, for a binary with no CFI protections, a ROP attack can target any of the code bytes, and hence has a 0% AIR value, whereas a CFI scheme

To ensure a like-for-like comparison with Lockdown [44], we report the AIR metric calculated dynamically on termination of the program, accounting only for executed indirect call targets by that point (figure 12). We separately calculate this statically to match BinCFI's calculation [59] (figure 13). In figure 12, we show two configurations for Lockdown; the default *strong* policy, *Lockdown (S)*, and a weaker policy, *Lockdown (W)*. Although the overall AIR metric of Lockdown's stronger policy is higher than JCFI[15], it suffers from unsoundness. The weaker policy mitigates the issues at the cost of a lower AIR metric. The AIR metric for backward edges is the same for JCFI and Lockdown as they both use a precise shadow-stack policy, but much higher than BinCFI since the latter allows returns to target to any call-preceded instruction. JCFI thus provides significantly better security compared to sound versions of the two comparisons.

6.3 Code Coverage in Janitizer

Finally, to understand the need for a hybrid security framework, figure 14 shows the fraction of basic blocks that are exclusively seen by the dynamic modifier, as they were either missed or not visible to the static analyzer. Bars can only be seen for benchmarks with more then 0.1% of basic block dynamically analyzed. On average, 4.4% of executed basic blocks only appear dynamically, with the ratio as high as 92.4% for cactusADM and 18.7% for lbm, coming from only two basic blocks in the latter case.

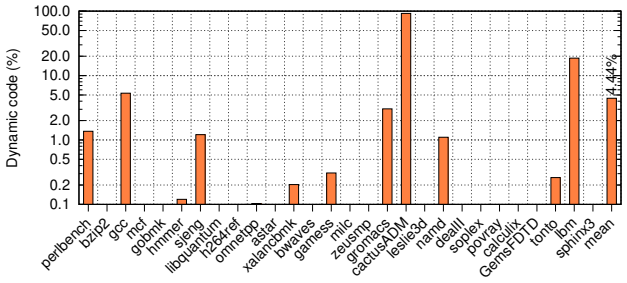


Figure 14. Fraction of basic blocks identified and analyzed dynamically (i.e. unseen in Janitizer's static analyzer).

7 Related Work

Development of defense and mitigation techniques against security exploits is a very active area of research. These techniques are either incorporated into hardware, kernel and compilers, or implemented to work with binaries. Among these, binary-based solutions that use frameworks that perform static binary rewriting or dynamic translation and modification receive significant interest.

Alto and Diablo [54] are link-time binary rewriting frameworks and have been employed in developing security techniques, such as instruction-set randomization [31]. However, they rely on the presence of debug information or binaries compiled only by specialized compilers, and are inapplicable to code not available at link-time. Retrowrite [23] has been used to implement memory sanitization and fuzzing techniques, but its scope is limited to position-independent binaries. REIL [24] is a reverse-engineering and static-analysis framework for ROP-gadget search. SecondWrite [5] facilitates the implementation of security policies, such as protection against stack overflows and control-flow hijacking [41], through binary rewriting, but faces challenges in handling large, complex binaries [23] and in effectively managing control-flow intricacies and dynamic code. StackArmor [19] uses static analysis for binary stack protection, suffering from typical limitations of static techniques in terms of coverage and protection of dynamic code.

Memcheck [48], built on top of Valgrind [40], is a widely used dynamic instrumentation tool for memory-safety related issues. However, it incurs very high memory and performance overhead [52] limiting its use to debugging rather than in production environments. Other dynamic translation and instrumentation frameworks, such as Pin [36] and DynamoRIO [14], have also been a popular choice to detect security vulnerabilities and build defenses against memory corruption and control-flow hijacking attacks [15, 20, 22]. Dr Memory [15], based on DynamoRIO, improves on memory and performance overhead compared to Memcheck, but still with high performance overheads up to 10×. Several CFI schemes [1, 2, 25] have been implemented using both static and dynamic techniques. ROPDefender [22], a dynamic tool built on Pin, enforces CFI for binaries, but has higher overhead than static techniques.

8 Conclusion

The instrumentation of binaries for security analysis has been well explored, but the sheer volume of tools and techniques necessary for high-performance, high-coverage, correct analysis, solely the purview of static-dynamic hybrid tools, has made watertight techniques a real challenge. With Janitizer, these barriers have now been diminished. We have shown how to build diverse techniques on top of Janitizer built on top of diverse static and dynamic code analysis, and hope that it will pave the way forward for many more.

Additional data related to this publication is available in the repository at https://doi.org/10.17863/CAM.113967. Janitizer is available at https://github.com/CompArchCam/Janus/tree/security.

aims to reduce the number of code bytes that can be be used as a potential target hence improves the AIR value.

[15]Despite higher net AIR value for Lockdown (S), its AIR for indirect jumps is lower than JCFI because Lockdown lets indirect jumps target any byte in the same function (identified by closest symbol), whereas JCFI restricts targets to instruction-/function boundaries via static control-flow analysis, hence why JCFI's AIR reduces from 99.8% to 99.6% without static analysis (hybrid compared to dynamic).

References

[1] 2016. Return Flow Guard. https://xlab.tencent.com/en/2016/11/02/return-flow-guard.

[2] 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 '05)*.

[3] Javed Absar. 2018. Scalar Evolution - Demystified. https://llvm.org/devmtg/2018-04/slides/Absar-ScalarEvolution.pdf.

[4] Sam Ainsworth and Timothy M. Jones. 2020. The Guardian Council: Parallel Programmable Hardware Security. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. https://doi.org/10.1145/3373376.3378463

[5] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. 2013. A Compiler-Level Intermediate Representation Based Binary Analysis and Rewriting System. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*. https://doi.org/10.1145/2465351.2465380

[6] ARM. 2009. *Building a Secure System using TrustZone Technology*. Technical Report. ARM Technical White Paper. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf

[7] Paul-Antoine Arras, Anastasios Andronidis, Luís Pina, Karolis Mituzas, Qianyi Shu, Daniel Grumberg, and Cristian Cadar. 2022. SaBRe: Load-Time Selective Binary Rewriting. *Int. J. Softw. Tools Technol. Transf.* 24, 2 (apr 2022), 205–223. https://doi.org/10.1007/s10009-021-00644-w

[8] Arash Baratloo, Navjot Singh, Timothy K Tsai, et al. 2000. Transparent Run-time Defense Against Stack-smashing Attacks.. In *USENIX Annual Technical Conference, General Track*. 251–262.

[9] Erick Bauman, Zhiqiang Lin, and Kevin W Hamlen. 2018. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics.. In *NDSS'18*.

[10] Andrew R. Bernat and Barton P. Miller. 2011. Anywhere, Any-time Binary Instrumentation. In *Proceedings of the 10th ACM Workshop on Program Analysis for Software Tools (PASTE'11)*. https://doi.org/10.1145/2024569.2024572

[11] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: A New Class of Code-reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS '11)*. https://doi.org/10.1145/1966913.1966919

[12] Tim Boland and Paul E Black. 2012. Juliet 1. 1 C/C++ and java test suite. *Computer* 45, 10 (2012), 88–90.

[13] Ferdinand Brasser, Lucas Davi, David Gens, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2017. Can't Touch this: Software-only Mitigation against Rowhammer Attacks Targeting Kernel Memory. In *Proceedings of the 26th USENIX Conference on Security Symposium (SEC'17)*.

[14] 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: Feedback-Directed and Runtime Optimization (CGO '03)*.

[15] Derek Bruening and Qin Zhao. 2011. Practical Memory Checking with Dr. Memory. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*.

[16] Derek Bruening and Qin Zhao. 2015. Building Dynamic Tools with DynamoRIO on x86 and ARM. https://www.burningcutlery.com/derek/docs/DynamoRIO-tutorial-2015.pdf.

[17] 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'11)*.

[18] Sanchuan Chen, Zhiqiang Lin, and Yinqian Zhang. 2021. SELECTIVETAINT: Efficient Data Flow Tracking With Static Binary Rewriting. In *30th USENIX Security Symposium (USENIX Security 21)*.

[19] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. 2015. StackArmor: Comprehensive Protection From Stack-based Memory Error Vulnerabilities for Binaries. In *NDSS'15*. https://doi.org/10.1109/SP40000.2020.00009

[20] W. Cheng, Qin Zhao, Bei Yu, and S. Hiroshige. 2006. TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting. In *11th IEEE Symposium on Computers and Communications (ISCC'06)*. https://doi.org/10.1109/ISCC.2006.158

[21] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. 2000. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In *Proceedings DARPA Information Survivability Conference and Exposition (DISCEX'00)*.

[22] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. 2011. ROPdefender: A Detection Tool to Defend against Return-oriented Programming Attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. https://doi.org/10.1145/1966913.1966920

[23] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *2020 IEEE Symposium on Security and Privacy (S&P'20)*. https://doi.org/10.1109/SP40000.2020.00009

[24] Thomas Dullien, Tim Kornau, and Ralf-Philipp Weinmann. 2010. A Framework for Automated Architecture-Independent Gadget Search. In *Proceedings of the 4th USENIX Conference on Offensive Technologies (WOOT'10)*.

[25] Ulfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. 2006. XFI: Software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*.

[26] Alan Eustace and Amitabh Srivastava. 1995. ATOM: A Flexible Interface for Building High Performance Program Analysis Tools. In *Proceedings of the USENIX 1995 Technical Conference Proceedings (TCON'95)*.

[27] William H. Hawkins, Jason D. Hiser, and Jack W. Davidson. 2016. Dynamic Canary Randomization for Improved Software Security. In *Proceedings of the 11th Annual Cyber and Information Security Research Conference (CISRC '16)*. https://doi.org/10.1145/2897795.2897803

[28] Christian Heitman and Iván Arce. 2014. BARF: a multiplatform open source binary analysis and reverse engineering framework. In *Congreso Argentino de Ciencias de la Computación*.

[29] John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News* 34, 4 (sep 2006).

[30] R. Nigel Horspool and Nenad Marovac. 1980. An Approach to the Problem of Detranslation of Computer Programs. *Comput. J.* 23, 3 (1980), 223–229.

[31] Wei Hu, Jason Hiser, Dan Williams, Adrian Filipi, Jack W Davidson, David Evans, John C Knight, Anh Nguyen-Tuong, and Jonathan Rowanhill. 2006. Secure and Practical Defense against Code-injection Attacks using Software Dynamic Translation. In *Proceedings of the 2nd international conference on Virtual Execution Environments (VEE'06)*. https://doi.org/10.1145/1134760.1134764

[32] Johannes Kinder. 2010. *Static analysis of x86 Executables*. Technical Report. Technische Universität Darmstadt.

[33] S. Krahmer. 2005. x86-64 Buffer Overflow Exploits and the Borrowed Code Chunks Exploitation technique. http://forum.ouah.org/nonx.pdf

[34] Mike Larkin. 2015. Kernel W^X Improvements In OpenBSD.

[35] Michael A Laurenzano, Mustafa M Tikir, Laura Carrington, and Allan Snavely. 2010. PEBIL: Efficient Static Binary Instrumentation for Linux. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. https://doi.org/10.1109/ISPASS.2010.5452024

[36] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. https://doi.org/10.1145/1065010.1065034

[37] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '13)*. https://doi.org/10.1145/2487726.2488368

[38] Xiaozhu Meng and Barton P. Miller. 2016. Binary Code is Not Easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. https://doi.org/10.1145/2931037.2931047

[39] Robert Muth, Saumya Debray, Scott Watterson, and Koen De Bosschere. 2001. Alto: A Link-time Optimizer for the Compaq Alpha. *Software: Practice and Experience* 31 (2001).

[40] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. https://doi.org/10.1145/1250734.1250746

[41] Pádraig O'Sullivan, Kapil Anand, Aparna Kotha, Matthew Smithson, Rajeev Barua, and Angelos D Keromytis. 2011. Retrofitting security in COTS software with Binary Rewriting. In *IFIP International Information Security Conference*.

[42] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'19)*.

[43] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. 2021. SoK: All You Ever wanted to Know about x86/x64 Binary Disassembly but were Afraid to Ask. In *2021 IEEE symposium on security and privacy (S&P'21)*.

[44] Mathias Payer, Antonio Barresi, and Thomas R Gross. 2014. Lockdown: Dynamic control-flow integrity. *arXiv preprint arXiv:1407.0549* (2014).

[45] Soumyakant Priyadarshan, Huan Nguyen, Rohit Chouhan, and R Sekar. 2023. {SAFER}: Efficient and {Error-Tolerant} Binary Instrumentation. In *32nd USENIX Security Symposium (USENIX Security 23)*.

[46] Soumyakant Priyadarshan, Huan Nguyen, and R Sekar. 2023. Accurate Disassembly of Complex Binaries Without Use of Compiler Metadata. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*. https://doi.org/10.1145/3623278.3624766

[47] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12)*.

[48] Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '05)*.

[49] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the Effectiveness of Address-space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS '04)*. https://doi.org/10.1145/1030083.1030124

[50] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Security and Privacy (S&P'16)*. https://doi.org/10.1109/SP.2016.17

[51] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *International Conference on Information Systems Security*. https://doi.org/10.1007/978-3-540-89862-7_1

[52] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: Sanitizing for Security. In *2019 IEEE Symposium on Security and Privacy (S&P'19)*. https://doi.org/10.1109/SP.2019.00010

[53] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical Context-Sensitive CFI *(CCS '15)*. 14 pages. https://doi.org/10.1145/2810103.2813673

[54] Ludo Van Put, Dominique Chanet, Bruno De Bus, Bjorn De Sutter, and Koen De Bosschere. 2005. Diablo: a Reliable, Retargetable and Extensible Link-time Rewriting Framework. In *International Symposium on Signal Processing and Information Technology*. https://doi.org/10.1109/ISSPIT.2005.1577061

[55] Cheng Wang, Shiliang Hu, Ho-seop Kim, Sreekumar R. Nair, Mauricio Breternitz, Zhiwei Ying, and Youfeng Wu. 2007. StarDBT: An Efficient Multi-platform Dynamic Binary Translation System. In *Advances in Computer Systems Architecture*.

[56] Shuai Wang, Pei Wang, and Dinghao Wu. 2015. Reassembleable disassembling. In *24th USENIX Security Symposium (SEC'15)*.

[57] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA '14)*. https://doi.org/10.1145/2678373.2665740

[58] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R Sekar. 2014. A Platform for Secure Static Binary Instrumentation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE'14)*. https://doi.org/10.1145/2576195.2576208

[59] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Conference on Security (SEC'13)*.

[60] Ruoyu Zhou and Timothy M. Jones. 2019. Janus: Statically-Driven and Profile-Guided Automatic Dynamic Binary Parallelisation. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'19)*. https://doi.org/10.1109/CGO.2019.8661196

[61] Ruoyu Zhou, George Wort, Márton Erdős, and Timothy M. Jones. 2019. The Janus Triad: Exploiting Parallelism through Dynamic Binary Modification. In *Proceedings of the 15th ACM International Conference on Virtual Execution Environments (VEE'19)*. https://doi.org/10.1145/3313808.3313812

Received 2024-09-12; accepted 2024-11-04