Retrofitting Intel CET's Indirect Branch Tracking into Legacy Binaries through Static Binary Rewriting

Bruno R. Ribeiro[1] , **Andrei Rimsa**[1] , **Mateus Tymburibá**[1] , **Eduardo P. Souto**[2]

[1]Departamento de Computação
Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG)
Belo Horizonte – MG – Brasil

[2]Instituto de Computação – Universidade Federal do Amazonas (UFAM)
Manaus – AM – Brasil

brunor.ribeiro96@gmail.com,{andrei,mateustymbu}@cefetmg.br,
esouto@icomp.ufam.edu.br

Abstract. *Modern processors increasingly embed hardware support for control-flow integrity. Intel CET's Indirect Branch Tracking (IBT) restricts indirect transfers to targets explicitly marked with* `endbr64`*. However, legacy binaries that cannot be recompiled remain outside this protection, leaving deployed software exposed to code-reuse attacks. This paper presents* BIN2CET*, an automated static rewriting system for retrofitting IBT support into legacy ELF binaries without requiring perfect control-flow recovery. The system combines* `endbr64`*-based target instrumentation, relocation-aware trampolines to preserve overwritten instructions, and conservative* `notrack` *handling for indirect branches that cannot be safely normalized. We evaluate* BIN2CET *on the 102 CoreUtils programs compiled under multiple optimization levels. The results show complete coverage of recovered function-entry targets in the non-optimized setting and above 80% coverage under optimized layouts. Runtime overhead ranges from approximately 22% to 40%, while attack-surface analysis shows that the policy-valid indirect-control-transfer surface is reduced to less than 1% of the syntactic gadget space. These results indicate that automated static rewriting is a practical path for bringing IBT compatibility to legacy binaries.*

1. Introduction

The evolution of cyber threats has been marked by the continuous refinement of techniques designed to bypass traditional defense mechanisms [Botacin et al. 2017]. This progression has motivated the deployment of increasingly stronger protections, including *Data Execution Prevention* (DEP), *Address Space Layout Randomization* (ASLR), and *Control-Flow Integrity* (CFI) [Nour et al. 2023]. Although these mechanisms have raised the bar for vulnerability exploitation, they have not eliminated code-reuse attacks. Techniques such as *Return-Oriented Programming* (ROP) and *Jump-Oriented Programming* (JOP) still allow attackers to reuse legitimate code fragments already present in a binary to divert program execution [Shacham 2007, Shanbhogue et al. 2019].

In response to this threat model, modern processors began to incorporate hardware-assisted mechanisms for enforcing control-flow integrity. Intel's *Control-Flow Enforcement Technology* (CET) is one such mechanism [Intel 2019]. CET includes two

complementary components: *Shadow Stack*, which protects return addresses using a protected stack separate from the regular execution stack, and *Indirect Branch Tracking* (IBT), which restricts indirect branches to targets explicitly marked as valid. In the case of IBT, valid indirect-branch destinations must start with the `endbr64` instruction. As a result, binaries compiled with CET-aware toolchains can expose a more constrained control-flow surface for indirect transfers.

Despite this architectural support, the adoption of IBT still faces a major practical obstacle: a large body of deployed software cannot be recompiled because its source code is unavailable, outdated, proprietary, or otherwise inaccessible. This situation is particularly relevant for components classified as *Software of Unknown Provenance* (SOUP), for which native recompilation with CET support is often not a viable option [Xie et al. 2022]. Such binaries typically do not contain the `endbr64` markers required by IBT. Consequently, systems that must continue executing these binaries may need to disable, bypass, or relax hardware-enforced protection, leaving legacy software outside the security benefits provided by modern processors.

Static binary rewriting offers a promising path for bridging this gap: it can transform compiled artifacts without requiring access to source code [Botacin et al. 2018]. However, retrofitting IBT into legacy binaries is technically challenging. The transformation must insert valid IBT landing pads while preserving the original program semantics. Naively injecting a 4-byte `endbr64` instruction into already compiled code may corrupt the code layout, violate instruction boundaries, break function alignment, or invalidate addressing modes that depend on the current instruction position, such as RIP-relative references. These challenges are amplified in optimized binaries, where function-entry layouts may be compact, irregular, or partially recovered.

A central difficulty is that complete and precise control-flow recovery is not a realistic assumption for arbitrary binaries. Indirect branches, compiler optimizations, stripped symbols, and mixed code/data regions can make it difficult to determine every possible control-flow target with certainty [Burow et al. 2017]. Nevertheless, practical IBT compatibility does not always require full recovery. Many indirect transfers target function entries or other structurally recoverable locations, while unresolved cases can be handled conservatively to preserve compatibility. This observation motivates an automated retrofitting system that combines target instrumentation, semantic preservation, and conservative fallback mechanisms instead of depending on perfect control-flow graph recovery.

This paper presents BIN2CET, an automated static binary rewriting system for retrofitting Intel CET's Indirect Branch Tracking into legacy ELF binaries. The system combines three complementary strategies. First, it instruments recovered indirect-branch targets with `endbr64` markers whenever the corresponding code region can be safely rewritten. Second, it preserves the semantics of overwritten instructions by moving them to auxiliary trampolines and correcting relocation-sensitive operands, including RIP-relative references. Third, it conservatively handles indirect branches that cannot be safely normalized by using `notrack` annotations. All together, these strategies allow BIN2CET to improve IBT compatibility without requiring source-code recompilation or perfect control-flow graph recovery.

The proposed system is designed to cover both favorable and less regular binary layouts. In optimized binaries, where many functions may not preserve canonical prologues, BIN2CET still attempts to instrument recoverable targets and relies on conservative handling when safe target-side rewriting is not possible. Therefore, uncovered targets and remaining runtime violations are treated as part of the empirical boundary of the approach, making the system suitable for studying the practical limits of IBT retrofitting across optimization levels.

We implement BIN2CET using LIEF [Thomas 2017] and an extended version of E9Patch [Duck et al. 2020]. LIEF is used for ELF-level manipulation, including binary annotation and direct instruction replacement, while E9Patch is extended to support relocation-aware trampolines and CET-compatible control transfers. We evaluate BIN2CET on 102 CoreUtils programs compiled as PIE binaries under multiple optimization levels. The results show complete coverage of recovered function-entry targets in the non-optimized setting and above 80% coverage under optimized layouts, with conservative handling used when safe target-side rewriting is not possible. We measure the security effect of this transformation as the reduction of the policy-valid indirect-transfer surface, namely the set of destinations that remain reachable through indirect control transfers under the rewritten IBT-oriented policy, rather than as the elimination of all syntactically identifiable gadgets.

2. Retrofitting Legacy Binaries for IBT Compatibility

Intel CET's Indirect Branch Tracking (IBT) enforces a simple but strict execution rule: the target of an indirect branch must be an instruction explicitly recognized as a valid landing pad, typically `endbr64`. This requirement is naturally satisfied when programs are compiled with CET-aware toolchains. However, legacy binaries compiled without IBT support do not contain these markers and, therefore, may fail when executed under an IBT-enforcing environment.

Retrofitting IBT support into such binaries requires more than adding `endbr64` instructions. A binary has a fixed code layout, and its instructions may rely on their original positions in memory. Rewriting a few bytes at a function entry can overwrite existing instructions, change instruction alignment, and invalidate instruction-pointer-relative operands. Therefore, a correct transformation must both introduce valid IBT landing pads and preserve the semantics of the original code.

Example 1 illustrates a common pattern in which a callback is implemented as an indirect function call.

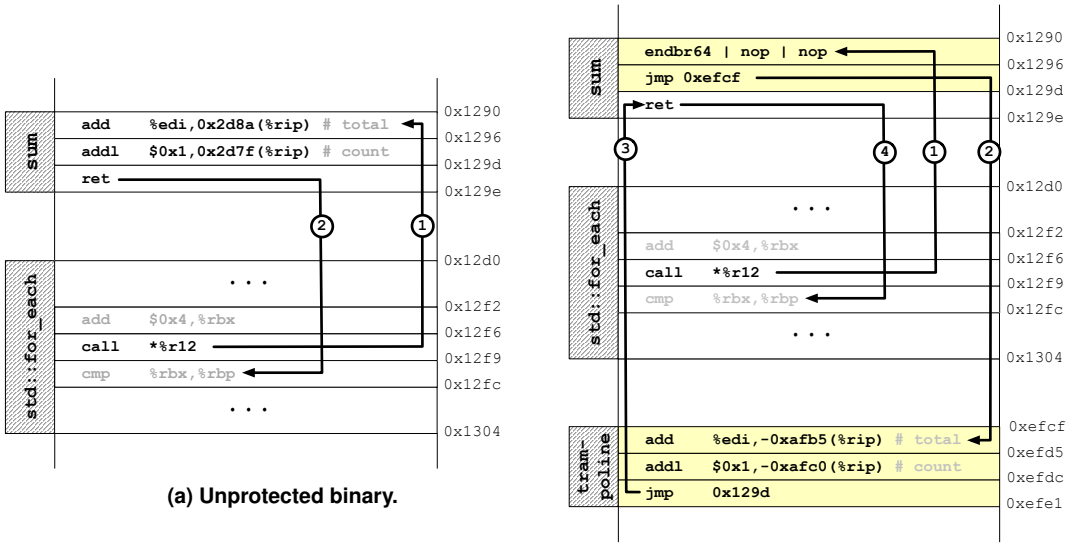
Example 1 *The C++ program in Figure 1 uses the standard C++ algorithm* `std::for_each` *to iterate over all elements of the* `std::vector<int> v`*. For each value, the* `std::for_each` *invokes function* `sum` *as a callback to count and accumulate the values passed as command-line arguments. Although this is a standard source-level programming pattern, the callback is implemented at binary level as an indirect call, whose destination must be valid under IBT.*

The program in Figure 1 can be used as an example of a legacy program that can be rewritten to support CET's IBT extension. First, we compile it without protection in Figure 2a. Then, we manipulate the unprotected binary replacing some of its instructions,

```
01 #include <cstdio>
02 #include <cstlib>
03 #include <algorithm>
04 #include <vector>
05 static int total = 0;
06 static int count = 0;
07 void sum(int n) {
08   total += n; count++;
09 }
10 int main(int argc, char* argv[]) {
11   std::vector<int> v;
12   for (int i = 1; i < argc; i++)
13     v.push_back(atoi(argv[i]));
14
15   std::for_each(v.begin(), v.end(), sum);
16   printf("%d in %d values\n", total, count);
17   return 0;
18 }
```

Figure 1. C++ program that invokes `sum` as a callback through `std::for_each`.

relocating overwritten instructions to trampolines, and adding explicit control transfers to preserve the original semantics in Figure 2b.



(a) Unprotected binary.

(b) Protected, CET-compatible, binary.

Figure 2. Instruction snippets for the program in Figure 1 before (a) and after (b) CET compatibility.

Definition 1 *To avoid imprecisions, in the text we will call a legacy program not CET-aware as "Unprotected", while a program that is retrofitted to be CET-compatible as "Protected". Thus, the code snippet from Figure 2a is considered "Unprotected", while the one from Figure 2b is considered "Protected".*

Example 2 *Figure 2a shows the instruction-level control flow of the program in Figure 1 when compiled without protection* [1]*. The callback is translated into x86_64 assembly as an indirect call instruction,* `call *%r12`*, located at address* `0x12f6`*. During normal execution, this instruction transfers control to function* `sum`*, at address* `0x1290` *(point 1). The program then executes* `sum` *and eventually reaches a return instruction that resumes execution in* `std::for_each`*, at address* `0x12f9` *(point 2). This execution is semantically valid for the original program, but it is not valid under IBT. Since the binary*

[1]The program in Figure 1 was compiled using `-fcf-protection=none`. It includes flags `-O2 -fno-stack-protector -fno-inline -fno-reorder-blocks-and-partition` to generate optimized code, but simple enough for illustration purposes.

was compiled without CET support, the target at address `0x1290` does not start with `endbr64`. Moreover, the indirect call itself is not annotated with `notrack`. Therefore, an IBT-enforcing environment would block the transfer to `sum`, even though the target is legitimate according to the program semantics. The example also exposes a second challenge. Function `sum` accesses the global variables `total` and `count`. The compiler emits instructions that access these variables through RIP-relative addressing, where the effective address is computed relative to the position of the instruction in the binary. For example, `total` is referenced at `0x1290` through `0x2d8a(%rip)`, while `count` is referenced at `0x1296` through `0x2d7f(%rip)`. Instructions of this kind require special handling during transformation, because moving them to a different location changes the base address used to compute the effective memory target.

There are two possible ways to make the indirect transfer compliant with IBT. The first is to annotate the indirect branch with `notrack`, which disables target tracking for that transfer. The second is to ensure that the branch target starts with `endbr64`. The former strategy is useful for compatibility, but it is less restrictive from a security perspective because it bypasses the target-side IBT check for that transfer. The latter strategy is preferable whenever it can be applied safely, because it makes the valid destination explicit and visible to the IBT policy. In practice, our system combines both strategies: it prioritizes `endbr64`-based target instrumentation and uses `notrack` conservatively when target-side rewriting is not safe or not possible.

Example 3 *Figure 2b shows the resulting protected binary. The highlighted instructions, in light yellow, correspond to the modifications introduced by the instrumentation. The key idea is to make the target of the indirect call at `0x12f6` start with an `endbr64` instruction (point 1), while preserving the behavior of the instructions originally located at that target. This is only possible if the target's basic block – a maximal sequence of instructions without branches, except at the end [Aho et al. 2006] – is longer than 4 bytes, the length of the `endbr64` instruction. In our example, the first instruction of this basic block has 6 bytes, enough space to be safely replaced by the `endbr64`. The remaining space was padded with two 1-byte `nop` instructions. Afterwards, the next instruction is replaced by a jump to an auxiliary trampoline (point 2). The trampoline contains the instructions overwritten by the `endbr64` landing pad and by the jump sequence. In our example, the trampoline includes the relocated instruction that updates `total`, at address `0xefcf`, and the relocated instruction that updates `count`, at address `0xefd5`. Since both instructions use RIP-relative addressing, they cannot be copied as-is, without modifications. Their displacements must be recomputed so that, from the trampoline's location, they still access the same global variables as in the original binary. This relocation step is crucial for preserving the data-flow semantics of the program. After executing the relocated instructions, the trampoline jumps back to the original function body at the first instruction that was not overwritten (point 3). From that point on, execution proceeds normally inside `sum` until it eventually reaches the return instruction (point 4) which resumes execution. Thus, the rewritten binary exposes an IBT-valid landing pad while preserving the control-flow and data-flow behavior of the original program.*

The transformation illustrated in Example 3 is the preferred case: the indirect target is made explicit through an `endbr64` landing pad, and the overwritten instructions are preserved through a relocation-aware trampoline. In favorable cases, the target

function of an indirect call may preserve the classical frame-pointer-based prologue that provides a predictable rewriting opportunity. The sequence `push %rbp` followed by `mov %rsp,%rbp` occupies 4 bytes, which can easily fit the `endbr64` instruction. The next instruction, immediately after the `endbr64`, must be patched with a redirect to the trampoline, implemented with a 32-bit relative jump. This jump is 5-bytes long: one byte for the `0xe9` jump opcode and four bytes for the relative displacement. In the worst case, if the patched instruction is only 1-byte long, the transformation can still encode the jump by replacing that byte with `0xe9`, the jump opcode, and use the following bytes as the displacement [Duck et al. 2020]. As a result, the target site of an indirect branch requires between 5 and 9 bytes in total, with the first 4 bytes reserved for `endbr64`. However, not all indirect target sites satisfy this size constraint. In such cases, the indirect branch may be marked with `notrack` as a conservative fallback mechanism.

The goal of the proposed retrofitting process is therefore not to recover or rewrite the entire control-flow graph. Instead, it aims to maximize the number of indirect transfers that can be normalized into explicit IBT-valid targets, while using conservative handling only when safe target-side instrumentation is not possible. This design enables practical IBT compatibility for legacy binaries while keeping the transformation local, semantics-preserving, and robust to partially recovered binary structure.

3. BIN2CET: An Automated IBT Retrofitting System

BIN2CET is an automated static rewriting system for retrofitting Intel CET's Indirect Branch Tracking (IBT) into legacy Linux ELF binaries, including executables and shared libraries. The system takes as input an unprotected binary and a set of candidate patch sites extracted from binary analysis. Each patch site contains the information required to drive the transformation, such as instruction addresses, section names and offsets, file offsets, instruction sizes, and branch-site metadata. This information may be manually provided by an analyst, automatically generated by an analyzer, or produced through a combination of both.

The design of BIN2CET follows the hybrid retrofitting strategy introduced in Section 2. Rather than requiring a perfect reconstruction of the program's control-flow graph, the system rewrites the parts of the binary that can be safely normalized and uses conservative handling for the remaining cases. More specifically, BIN2CET supports three transformation strategies:

1. Indirect branch target: Replaces a recovered indirect-branch target with an `endbr64` landing pad and redirects execution to a trampoline containing the overwritten instructions, possibly after relocation.
2. Indirect call: annotates an indirect call with `notrack` when target-side instrumentation is not safe or not applicable.
3. Indirect jump: applies the same `notrack`-based handling to indirect jump instructions.

The first strategy is preferred whenever the target region can be safely rewritten, because it makes the destination explicit and visible to the IBT policy. The second and third strategies are fallback mechanisms used to preserve compatibility when an indirect transfer cannot be normalized through target-side instrumentation. Although indirect calls and indirect jumps are handled similarly at the instruction level, BIN2CET treats them as

separate strategies so that the user can enable one, both, or neither, depending on the desired trade-off between compatibility and strictness.

The rewriting workflow is organized into three stages: binary analysis, binary transformation, and runtime validation. First, the input legacy ELF binary is analyzed to recover candidate function entries, indirect control-transfer instructions, and relocation-sensitive code regions. The analysis stage produces a patch-site description, which serves as interface between the analyzer and the rewriting strategies. Next the transformation stage applies the selected rewriting strategies and emits a rewritten IBT-compatible binary. Finally, the runtime validation stage checks whether observed indirect branches reach valid `endbr64` targets or are explicitly handled with `notrack`. These validation results are later used in the experimental evaluation.

3.1. Binary Transformation

The binary transformation stage combines LIEF [Thomas 2017] (*Library to Instrument Executable Formats*) and an extended version of E9Patch [Duck et al. 2020]. LIEF performs direct ELF-level manipulation, including byte-level instruction replacement, padding, and insertion of binary annotations, while E9Patch materializes out-of-line trampolines and redirects control flow to and from them.

When an indirect-branch target is selected for instrumentation, BIN2CET uses LIEF to replace the corresponding target region with an `endbr64` instruction, adding padding when the replaced instruction sequence is larger than the landing pad. As discussed in Example 3, this replacement must respect instruction boundaries and leave enough space for the subsequent redirection to a trampoline. LIEF is also used to add the ELF metadata required to request CET-related protection at runtime. This information is represented through the `.note.gnu.property` section and can be inspected with `readelf`[2] using the `-n` option, as shown in Figure 3. In this work, the annotation declares the relevant x86 CET features, while the rewriting strategy and evaluation focus on IBT compatibility.

Figure 3. `readelf` snippet with added `.note.gnu.property` annotation.

```
Displaying notes found in: .note.gnu.property
  Owner                Data size        Description
  GNU                  0x00000010       NT_GNU_PROPERTY_TYPE_0
      Properties: x86 feature: IBT, SHSTK
```

E9Patch handles the out-of-line rewriting required after target instrumentation. We extend it in three ways: to insert `notrack` annotations when patching indirect calls and jumps, to relocate instructions overwritten by `endbr64`-based target instrumentation, and to adapt trampoline management so that the inserted control transfers remain compatible with the IBT-oriented rewriting policy. This relocation support is essential because instructions moved to a trampoline may depend on their original position in memory, as in RIP-relative memory references. When such instructions are relocated, BIN2CET recomputes their displacements so that they still refer to the same semantic targets accessed in the original binary. Without this step, the transformed binary could preserve control flow while silently corrupting data accesses.

[2] `readelf` is a tool for displaying ELF format information. For details, see `https://www.gnu.org/software/binutils/`.

3.2. Analysis Support

Before applying transformations, BIN2CET obtains candidate function entries, indirect calls, indirect jumps, instruction sizes, and relocation-sensitive instructions from the input binary. To support different scenarios, we provide two analyzers: one based on `objdump`[3] and another based on Ghidra [Nance and Eagle 2026].

The `objdump`-based analyzer is lightweight and fast, making it suitable when symbols and disassembly information are sufficient to identify patch sites. The Ghidra-based analyzer is slower but provides richer recovery when binaries are stripped or less regular. Both analyzers emit the same patch-site description consumed by BIN2CET, decoupling analysis from transformation and allowing improved analyzers to be integrated without changing the rewriting engine.

3.3. Runtime Validation

After rewriting, BIN2CET uses lightweight runtime checkers to validate observed IBT behavior. The checkers monitor indirect calls and jumps during execution and verify whether each observed target starts with `endbr64` or whether the transfer is explicitly handled through `notrack`. Although Intel SDE provides broader CET emulation [Intel 2026], our evaluation uses a PIN-based checker tailored to the IBT behavior exercised by the workloads considered in Section 4.

Overall, BIN2CET operationalizes the hybrid strategy described in Section 2: it makes recovered indirect targets explicit through `endbr64`, preserves overwritten instructions through relocation-aware trampolines, and uses `notrack` only as a conservative fallback

4. Experiments Methodology

The goal of making legacy binaries CET compatible is to ensure every indirect branch target always lands on an `endbr64` instruction. However, no algorithm is capable of determining with certainty every possible flow in a program. This problem is undecidable for the general case according to Rice's Theorem [Rice 1953]. To support a wide range of programs despite this limitation, we rely on the observation that most indirect calls target function entries. Therefore, the focus of our work is to identify all function entries in order to patch them with an `endbr64` instruction. For indirect jumps that cannot be normalized through target-side instrumentation, BIN2CET applies `notrack` as a conservative fallback. Indirect calls are primarily handled through their recovered function-entry targets: when the target function can be safely rewritten, the call remains subject to the target-side IBT check through `endbr64`. Explicit `notrack` instrumentation for indirect call sites is supported and was applied in selected cases, but most indirect calls in our evaluation were handled through target-side instrumentation rather than call-site rewriting.

The empirical study in this paper focuses on binaries from the CoreUtils suite. This benchmark family was selected because it provides a diverse collection of widely used command-line programs. This allows the proposed instrumentation to be evaluated

[3] `objdump` is a tool for displaying object-file information. For details, see `https://www.gnu.org/software/binutils/`.

across executables with distinct control-flow structures, code layouts, and runtime behaviors. To evaluate our solution, we use the entire CoreUtils suite comprised of 102 programs in a Docker container using the following workflow with the most common optimization levels: `-O0`, `-O2` and `-O3`.

Step 1: We compile every program without CET protection using gcc's `-fcf-protection=none` compiler flag. The resulting binaries are all Position Independent Executable (PIE). Although our solution supports non-PIE programs, we focus on PIE-only since `E9Patch` is more effective on this type of programs [Duck et al. 2020].

Step 2: We extract function entries, indirect calls and jumps using the faster `objdump` analyzer on the insecure binaries. This analyzer yields similar results when compared to the `GHidra` analyzer when applied to the CoreUtils programs.

Step 3: We feed the extracted information to BIN2CET to patch the binary in order to make it CET compatible. Not all function entries may be patched at this point: the initial basic block may not have enough space to hold the `endbr64`. This is not a problem, unless these functions are used in an indirect branch. In such cases, the indirect branch itself may be patched with the `notrack` marker. However, in most cases, compilers emit the classical function prologue that can easily fit the `endbr64` instruction.

Step 4: We then execute CoreUtils' testsuite on all 102 patched binaries to verify that the transformation is correct and does not break during execution. Then, we verify conformity with CET using our runtime checker. To support this, we create wrappers involving the binaries to execute with this PIN tool. Some of CoreUtils tests and 4 of its 102 programs (`timeout`, `env`, `nice`, and `stty`) were removed from this part of the evaluation due to incompatibilities with PIN. Finally, we extract runtime information with `perf` using a similar wrapper. We collect the following hardware counters: `instructions`, `cycles, task-clock`, `stalled-cycles-frontend`, `branch-misses`, `branches`, `cache-misses`, and `cache-references`.

5. Evaluation and Results

The goal of this section is to evaluate CET compatibility when rewriting binaries. To this end, we analyze the following research questions:

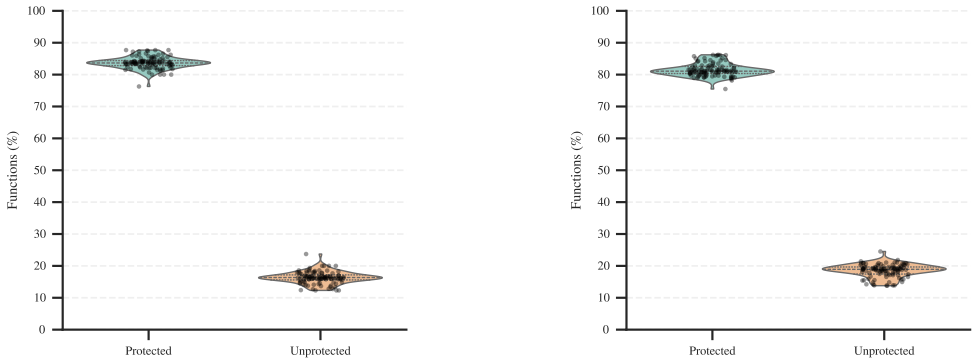
RQ1: How effective is BIN2CET to ensure CET compatibility?
RQ2: What are the runtime and binary-size overheads introduced by the rewriting process?
RQ3: How does BIN2CET affect the effective attack surface available to code-reuse attacks?

5.1. CET Compatibility Effectiveness

As described in Step 2 of the experiments methodology (§4), we use the `objdump` analyzer to extract function entries, indirect calls, and indirect jumps from the input binaries. The evaluated configuration focuses on recovered function entries and indirect jumps. This design follows the observation that most indirect calls target function entries; therefore, indirect calls are primarily handled through target-side instrumentation, avoiding unnecessary call-site rewriting when the target function can be protected with `endbr64`.

BIN2CET protected all 19,766 recovered function entries in the `-O0` configuration. This result is explained by the presence of classical function prologues, which usually provide enough space for safe `endbr64` insertion and trampoline-based relocation. In optimized binaries, function-entry coverage decreased to 84.16% in `-O2` and 82.17% in `-O3`, reflecting compact layouts that may prevent safe target-side rewriting. Figure 4 summarizes the percentage of function entries that BIN2CET was able to protect across optimization levels. The missing function entries are mainly due to the lack of space in the corresponding basic block or to E9Patch's inability to find a suitable location for the trampoline.

It is important to distinguish function-entry coverage from indirect branch-site handling. In `-O0`, BIN2CET patched all 7,046 extracted indirect jumps. In optimized binaries, indirect jumps were almost entirely handled: 7,730 out of 7,734 in `-O2` and 7,735 out of 7,735 in `-O3`. Indirect calls followed a different strategy: they were primarily handled through their recovered targets, while explicit call-site `notrack` instrumentation was applied only in selected cases. Thus, the reported 84.16% and 82.17% values measure recovered function-entry coverage, not the fraction of all indirect branch sites left untreated.



(a) Optimization level -O2. **(b) Optimization level -O3.**

Figure 4. Proportion in % of functions protected and unprotected in optimization level -O2 (a) and -O3 (b) for all of the 102 programs of CoreUtils.

Afterwards, we ran the runtime checker to verify whether observed indirect transfers still reached targets without `endbr64` or without an explicit `notrack` marker. No runtime IBT violations were observed in `-O0`. In `-O2` and `-O3`, violations were observed in `false` and `true`, their `main` function was invoked using an unprotected call from the C library and there was no suitable space to include the `endbr64` marker in this function prelude. Patching the call with `notrack` in the C library itself could mitigate this issue, but it was outside the scope of our evaluation. A different limitation appeared only in `-O2`, in `sha224sum`, `sha256sum`, `sha384sum`, and `sha512sum`: one extracted indirect jump could not be rewritten because no trampoline placement was found in a suitable address. All other programs, except for the ones that could not be evaluated with PIN (§4), had no observed runtime violations.

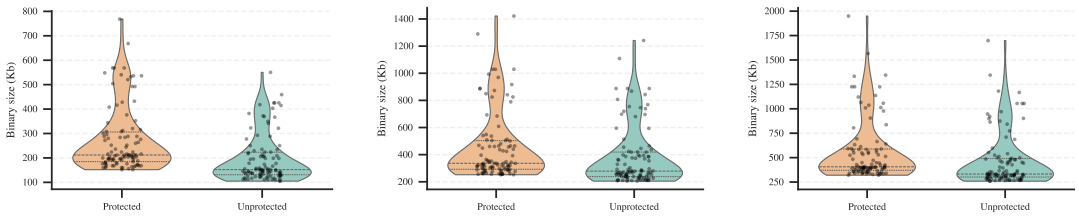
5.2. Runtime Overhead and Impact on Binary Size

To evaluate the impact of the instrumentation added by BIN2CET, we run perf to collect statistics for the programs in CoreUtils test suite, excluding four as mentioned in Section 4. Table 1 shows the weighted average based on the number of instructions executed per programs for 7 perf counters. We use counters `cycles` and `task-clock` to measure the runtime overhead: ∼22% for optimization level -O0, and ∼40% for -O2 and -O3. The impact of the instrumentation was high for CoreUtils, due to its nature of concise programs that run for few cycles and calls many small functions. Results may vary on more CPU intensive benchmarks, such as SPEC CPU. There were also spikes in the number of stalled cycles, branch and cache references, and consequently in cache-misses and cache-references for the increased executed instructions caused by the instrumentation.

Table 1. Average weighted overhead, based on number of instructions executed per program, for perf counters.

Perf Counter	Optimization -O0			Optimization -O2			Optimization -O3		
	Unprotected	Protected	Overhead	Unprotected	Protected	Overhead	Unprotected	Protected	Overhead
cycles	2265369.63	2782301.94	22.82%	2123523.84	3009095.28	41.70%	2128542.73	3015268.40	41.66%
task-clock	0.81	0.99	22.22%	0.76	1.07	40.79%	0.76	1.07	40.79%
stalled-cycles-frontend	1630736.98	1961997.33	20.31%	1526466.04	2067297.74	35.43%	1530630.42	2071754.38	35.35%
branch-misses	10464.15	13315.61	27.25%	10189.18	14660.20	43.88%	10224.57	14707.28	43.84%
branches	244650.31	307009.47	25.49%	229663.56	349808.99	52.31%	230221.44	349835.22	51.96%
cache-misses	4881.27	5190.76	6.34%	4383.11	4940.14	12.71%	4389.53	4937.34	12.48%
cache-references	29060.60	33540.06	15.41%	27512.60	32725.60	18.95%	27516.50	32733.89	18.96%

Figure 5, reports the binary size distribution before and after instrumentation for the three optimization levels. As expected, the rewriting process increases the binary size in all configurations, since it preserves displaced instructions, materializes control-flow redirections and inserts the branch-entry markers required by the policy. For optimization level `-O0`, the geometric mean increases from ∼178.4Kb to ∼250.5Kb, corresponding to an approximate 40.4% increase. For `-O2`, the geometric mean increases from ∼338.0Kb to ∼407.4Kb, while for `-O3` from ∼414.3Kb to ∼502.8Kb. These results show that the absolute size of optimized binaries is larger, but the relative growth introduced by instrumentation is more pronounced in the not optimized case.



(a) Optimization -O0. **(b) Optimization -O2.** **(c) Optimization -O3.**

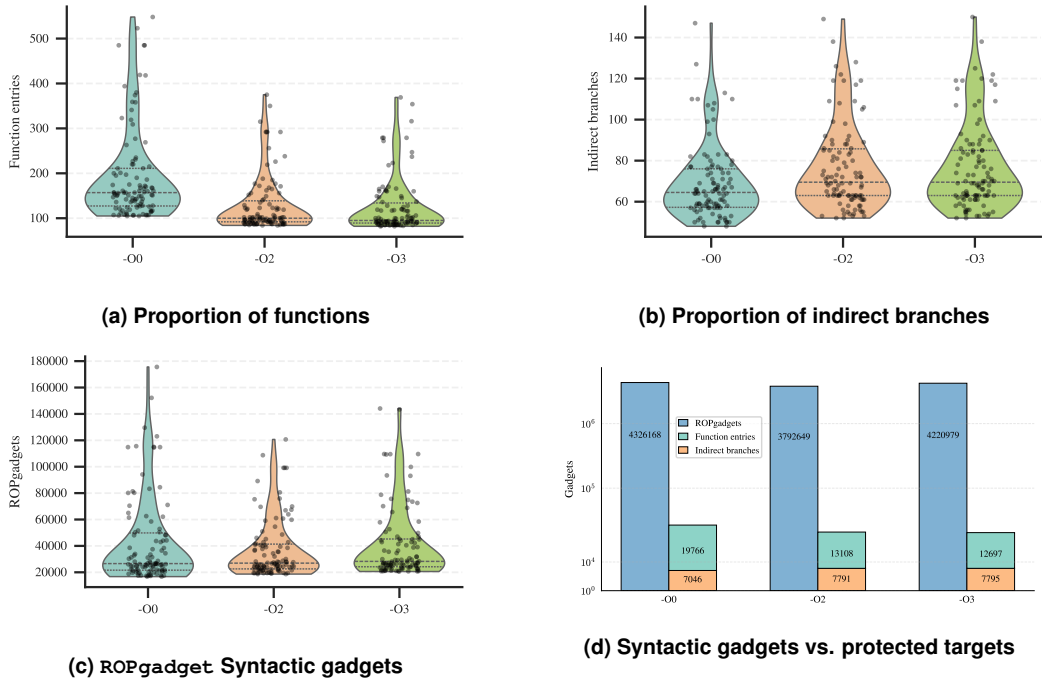
Figure 5. Binary-size distribution for all the 102 CoreUtils programs before and after instrumentation under `-O0` , `-O2`, `-O3`. The noted values indicate the geometric mean size in Kb.

5.3. Attack Surface Reduction

We also analyzed the effective attack surface available after rewriting. ROPgadget was executed with an aggressive configuration, using `--depth 100` and `--multibr`, without additional filtering options. This setup intentionally favors a broad syntactic enumeration of gadget-like instruction sequences. As a result, rewritten binaries may expose more

syntactically recognizable gadgets than the originals, since the transformation introduces trampolines, relocated instruction sequences, and auxiliary branch-handling blocks.

This increase should not be interpreted as a direct increase in the effective exploitation surface under the proposed policy. ROPgadget reports syntactic gadget patterns, but does not determine whether those sequences remain valid destinations under an IBT-oriented enforcement model. In contrast, our metric focuses on the destinations intentionally preserved by the transformation: `endbr64`-marked targets and conservatively preserved `notrack` sites.



(a) Proportion of functions

(b) Proportion of indirect branches

(c) `ROPgadget` Syntactic gadgets

(d) Syntactic gadgets vs. protected targets

Figure 6. Figures a and b show per-binary distributions of extracted function entries and indirect branch sites, respectively. Figure c reports the distribution of syntactic gadgets identified by `ROPgadget`. Figure d summarizes the aggregate gap between the syntactic gadget space and the protected targets preserved by the instrumentation policy (in logarithmic scale).

Figure 6 makes this distinction explicit across different optimization levels. Figures 6a and 6b show the distributions of recovered function entries and indirect branches, respectively, with indication that the control-transfer structures available for instrumentation remain substantial across `-O0`, `-O2`, and `-O3`. Figure 6c shows that the syntactic gadget space identified by `ROPgadget` remains broad in all three configurations, which is expected under an intentionally enumeration strategy. In contrast, Figure 6d shows that the protected target set preserved after instrumentation is only a small fraction of that syntactic space. The results indicate an approximate 99.2% reduction in the effective control-transfer surface, which corresponds only about 0.8% of the original syntactic gadget space. The security effect of the proposed rewriting strategy therefore lies in reducing the effective set of reachable indirect-branch destinations, rather than in removing every sequence that may be syntactically classified as a gadget.

This distinction is important for interpreting the security impact of the approach.

The proposed technique does not attempt to eliminate every syntactically recognizable gadget sequence from the binary. Rather, it restricts the destinations that can be reached through indirect branches to a smaller and explicitly managed set of branch-entry points. This interpretation is consistent with the observation by Schwartz et al. [Schwartz et al. 2011] that automated ROP construction depends on the effective density and availability of useful gadgets to the attacker. Under this view, the relevant result is not the raw number of sequences reported by a syntactic gadget finder, but the separation between syntactic gadget availability and the policy-level control-flow surface preserved.

6. Related Work

Static binary rewriting has been widely investigated as a means of modifying already compiled programs without access to their source code [Schulte et al. 2022]. In security, this capability is especially relevant for adapting legacy executables, inserting protection mechanisms, and patching binaries that are already deployed [Zhang et al. 2014, Hawkins et al. 2017, Duck et al. 2020]. Unlike dynamic instrumentation, static rewriting produces a transformed executable ahead of execution, avoiding the continuous runtime cost imposed by tools such as Pin and Valgrind [Bernat and Miller 2011, D'Elia et al. 2022]. However, rewriting binaries correctly remains difficult because the process depends on reliable disassembly, partial recovery of program structure, safe code transformation, and preservation of position-dependent instructions [Meng and Miller 2016, Duck et al. 2020].

The importance of these techniques has grown with the evolution of code-reuse attacks. Defenses such as *Data Execution Prevention* (DEP) and *Address Space Layout Randomization* (ASLR) raised the difficulty of exploiting memory-safety vulnerabilities, but did not eliminate attacks that reuse existing code fragments, including *Return-Oriented Programming* (ROP), *Call-Oriented Programming* (COP), and *Jump-Oriented Programming* (JOP) [One 1996, Shacham 2007, Shanbhogue et al. 2019]. In response, *Control-Flow Integrity* (CFI) mechanisms were proposed to restrict execution to valid control-flow targets [Zhang and Sekar 2013]. Their effectiveness, however, depends on the precision of the recovered control-flow information and on the strictness of the enforced policy, which remains challenging for stripped, optimized, or otherwise irregular binaries.

Intel's Control-Flow Enforcement Technology (CET) provides hardware support for strengthening control-flow integrity [Intel 2019, Xie et al. 2022]. Among its mechanisms, *Indirect Branch Tracking* (IBT) validates indirect transfers by requiring targets to start with an instruction such as `endbr64`, while *Shadow Stack* protects return addresses through a protected stack separated from the regular execution stack [Intel 2019]. For legacy binaries, IBT is a more localized retrofitting target than *Shadow Stack* because it focuses on valid indirect-branch destinations and selected branch sites rather than on all call/return behavior. Recent work such as FineIBT shows the security potential of constraining indirect-branch targets more precisely, but assumes environments in which the required instrumentation and policies can be made available at compile time or through stronger program knowledge [Gaidis et al. 2023].

The main gap addressed by this work lies at the intersection of static rewriting and hardware-enforced IBT compatibility for legacy software. Existing static rewriters

provide mechanisms for transforming binaries, but robustly inserting IBT landing pads requires preserving overwritten instructions and correctly relocating operands that depend on the original program counter, such as RIP-relative references [Duck et al. 2020, Smithson et al. 2013, Meng and Liu 2016]. Rather than pursuing full control-flow recovery, our approach targets recoverable function entries and indirect branch sites, uses relocation-aware trampolines to preserve semantics, and applies `notrack` conservatively when target-side rewriting is not safe. This positions our work as a practical static retrofitting strategy for improving IBT compatibility in legacy binaries while explicitly addressing the code-relocation challenges that make this problem difficult.

7. Conclusion and Future Work

This work presented BIN2CET, a static binary rewriting system for retrofitting Intel CET's IBT into legacy binaries without requiring source code. The system combines recovery of relevant control-flow targets, `endbr64`-based instrumentation, relocation-aware trampolines, and conservative `notrack` handling. By exploiting favorable binary structures when available and falling back to conservative handling when target-side rewriting is not safe, BIN2CET introduces IBT-compatible behavior while preserving the semantics of the original executable.

Our CoreUtils evaluation shows that the approach is practically viable. BIN2CET covered all recovered function-entry targets in the non-optimized configuration and achieved high coverage under optimized layouts, despite compact entry regions and trampoline-placement constraints. The transformed binaries executed successfully under the evaluated workloads, providing evidence that the rewriting process preserves functional behavior. The evaluation artifact is available at `https://doi.org/10.5281/zenodo.20420793` to support independent verification of these results. The development repository is also publicly available at `https://github.com/brunorrib/bin2cet`.

From a security perspective, BIN2CET reduces the effective space available for code-reuse attacks by constraining usable indirect-branch destinations to explicitly managed targets, rather than attempting to remove every syntactic gadget. The `perf`-based measurements show measurable but bounded overhead, while binary-size growth reflects the expected structural cost of inserting markers, materializing redirections, and preserving relocated instructions.

As future work, we plan to evaluate BIN2CET on more diverse binaries and production-oriented software stacks, such as Nginx linked against OpenSSL. We also intend to improve support for stripped binaries, reduce conservative `notrack` markings, improve relocation handling in complex layouts, and compare BIN2CET with other binary hardening approaches.

AI Usage Statement

During the preparation of this work, the authors used the Gemini tool (Google) exclusively to review grammar, improve clarity, and refine the writing style of the text. After using the tool, the authors carefully reviewed the document and assume full and complete responsibility for the final submitted content.

References

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, Boston, Massachusetts, USA.

Bernat, A. R. and Miller, B. P. (2011). Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, Madison.

Botacin, M., da Rocha, V. F., de Geus, P. L., and Grégio, A. (2017). Analysis, anti-analysis, anti-anti-analysis: an overview of the evasive malware scenario. *Simpósio Brasileiro de Cibersegurança (SBSeg)*, pages 250–263.

Botacin, M., Geus, P. L. D., and Grégio, A. (2018). Who watches the watchmen: A security-focused review on current state-of-the-art techniques, tools, and methods for systems and binary analysis on modern platforms. *ACM Computing Surveys (CSUR)*, 51(4):1–34.

Burow, N., Carr, S. A., Nash, J., Larsen, P., Franz, M., Brunthaler, S., and Payer, M. (2017). Control-flow integrity: Precision, security, and performance. *ACM Computing Surveys (CSUR)*, 50(1):1–33.

D'Elia, D. C., Invidia, L., Palmaro, F., and Querzoni, L. (2022). Evaluating dynamic binary instrumentation systems for conspicuous features and artifacts. *Digital Threats: Research and Practice*, 3(2).

Duck, G. J., Gao, X., and Roychoudhury, A. (2020). Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation*, Singapore.

Gaidis, A. J., Moreira, J., Sun, K., Milburn, A., Atlidakis, V., and Kemerlis, V. P. (2023). Fineibt: Fine-grain control-flow enforcement with indirect branch tracking. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*.

Hawkins, W. H., Hiser, J. D., Co, M., Nguyen-Tuong, A., and Davidson, J. W. (2017). Zipr: Efficient static binary rewriting for security. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

Intel (2019). Control-flow enforcement technology specification. Technical report, Intel. Acesso em: 20 nov. 2024.

Intel (2026). Intel® software development emulator (intel® sde). Accessed: 2026-05-23.

Meng, X. and Liu, W. (2016). Incremental cfg patching for binary rewriting. In *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*.

Meng, X. and Miller, B. P. (2016). Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*.

Nance, K. and Eagle, C. (2026). *The Ghidra Book, 2nd Edition: The Definitive Guide*. No Starch Press.

Nour, B., Pourzandi, M., and Debbabi, M. (2023). A survey on threat hunting in enterprise networks. *IEEE Communications Surveys & Tutorials*, 25(4):2299–2324.

One, A. (1996). Smashing the stack for fun and profit. *Phrack Magazine*, 7(49):14–16.

Rice, H. G. (1953). Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(1):358–366.

Schulte, E., Brown, M. D., and Folts, V. (2022). A broad comparative evaluation of x86-64 binary rewriters. In *Proceedings of the 15th Workshop on Cyber Security Experimentation and Test*.

Schwartz, E. J., Avgerinos, T., and Brumley, D. (2011). Q: Exploit hardening made easy. In *Proceedings of the 20th USENIX Security Symposium*. USENIX Association.

Shacham, H. (2007). The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 552–561.

Shanbhogue, V., Gupta, D., and Sahita, R. (2019). Security analysis of processor instruction set architecture for enforcing control-flow integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, New York, NY, USA. ACM.

Smithson, M., Elwazeer, K., Anand, K., Kotha, A., and Barua, R. (2013). Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness. In *20th Working Conference on Reverse Engineering (WCRE)*.

Thomas, R. (2017). Lief - library to instrument executable formats. https://lief.quarkslab.com/.

Xie, M., Wu, C., Zhang, Y., Xu, J., Lai, Y., Kang, Y., Wang, W., and Wang, Z. (2022). Cetis: Retrofitting intel cet for generic and efficient intra-process memory isolation. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, Beijing.

Zhang, M., Qiao, R., Hasabnis, N., and Sekar, R. (2014). A platform for secure static binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*.

Zhang, M. and Sekar, R. (2013). Control flow integrity for cots binaries. In *USENIX Security Symposium*.