Introduction

Program instrumentation plays a central role in software security, serving as the foundation for exploit mitigation, security policy enforcement, fuzzing, performance monitoring, and so on. Binary instrumentation is most widely applicable since it operates directly on software deployed on end systems. In addition to proprietary software that may only be available in binary form, most Linux software is distributed in binary form, thus making binary instrumentation very attractive even for open-source software. These factors have prompted a great deal of research in binary analysis and instrumentation techniques in recent years [3, 5, 12, 14, 17, 29, 41, 50, 51, 54, 55, 57, 58, 62].

The absence of symbolic information in deployed binaries makes their instrumentation very challenging. This problem is particularly formidable for the ubiquitous x86/x64 architecture due to its variable length instructions, the indistinguishability of data from code, and the difficulty of determining (indirect) control-flow targets. As a result, even the basic step of disassembly is error-prone: as per a recent survey [36], the best disassembly techniques have error rates in the range of 0.1% across benchmark suites, with some binaries leading to error rates in the 1% to 15% range. Instrumentation based on erroneous disassembly will naturally result in faulty programs.

The second major challenge is accurate code pointer identification. Note that instrumentation involves the insertion of code snippets. Existing instructions have to be moved around to create the space for this new code, and hence code pointer constants used in the original program will no longer point to their intended targets. Identifying and “fixing up” these pointers is an error-prone task because there is no reliable way to distinguish between integer and pointer-valued constants appearing in data or code. Modifying an integer value to “fix it up” — a false positive in code pointer identification — will change program behavior. Missing a code pointer fix-up — a false negative — will result in a control transfer to an unintended target, and hence cause the program to crash or misbehave.

Existing approaches for addressing these challenges fall into two broad groups: (I) robust, but slow, and (II) efficient but best-effort. Group I includes dynamic binary instrumentation (DBI) techniques [6, 7, 32] as well as BinCFI [62] and Multiverse [4] that prioritize compatibility. In particular, DBI techniques disassemble basic blocks before their first execution, thus side-stepping the challenges of static disassembly. Multiverse, a static binary instrumentation system, introduces superset assembly, which considers every byte as a possible beginning of an instruction. All of the recovered instructions are instrumented, which ensures that no code is missed. Techniques in Group I do not require reliable code pointer identification since they don’t change them at all. Instead, they go through runtime address translation that maps code locations in the original code to the corresponding locations in the instrumented code. Address translation is great for compatibility, but comes at a high performance cost since it needs to be performed on all indirect control transfers, including function returns.

Group II includes CCFIR [58] as well as the more recent Egalito [57], RETRORITE [17] and SBR [41] that achieve very good performance but make optimistic assumptions about the binary. Specifically, the last three works assume that:

*This work was supported by an ONR grant N00014-17-1-2891 and in part by NSF grants 1918667 and 2153056.
We now outline our approach and summarize its contributions. We answer this question in the affirmative and present our approach.

With these assumptions, they statically transform all code pointers so that they point to their intended new locations in the instrumented code. Consequently, they can avoid the costly address translation step, and achieve close to zero overhead. The above discussion raises the following research question:

*Can we combine the benefits of Groups I and II? That is, can binary instrumentation be very efficient, yet tolerate disassembly and code pointer identification errors?*

We answer this question in the affirmative and present our SAFer and Efficient binary Rewriter (SAFER) in this paper. Our system, along with the datasets and measurement procedures used, is available at [http://seclab.cs.sunysb.edu/soumyakant/safer](http://seclab.cs.sunysb.edu/soumyakant/safer). We now outline our approach and summarize its contributions.

### 1.1 Goals

Our approach operates on stripped binaries lacking symbols or any other compiler metadata 1 and has the following goals:

1. **Tolerate the most common disassembly and code pointer identification errors.** Tolerance means that these errors won’t affect the functionality of the instrumented code, but may add some runtime overhead.
2. **Detect the remaining disassembly/code pointer errors before they cause execution divergence.** Undetected errors can propagate, leading to undefined behaviors that can result in serious damage, e.g., security attacks or data loss. In contrast, early detection can provide a fail-crash primitive, and/or serve as a trigger that can be used to initiate an orderly shutdown or recovery.
3. **Incur low overhead.** We aim to achieve performance overheads close to 2% on average.
4. **Be compatible with complex code.** Instrumentation should be transparent, and work with many features of complex code, including the uses of (a) position-dependent executables (non-PIEs), (b) data embedded within code, (c) C++ exceptions and stack unwinding.

About 5% of the binaries found on our experimental platform (Ubuntu 20.04) are non-PIE, with many developer tools (e.g., gcc, llvm, gcov) continuing in this format even in more recent versions. Non-PIE is the norm for legacy code. It is also more common for custom/proprietary software to be non-PIE.

While data in the midst of code is unusual, many high-profile applications (e.g., ssh, apt, gimp, evince and firefox) use at least one such binary (e.g., libgcrypt.so or libxul.so). Among the real-world applications considered in our evaluation, most of the larger ones use one or more such libraries. Security applications require all code to be instrumented, so handling such binaries is important.

Finally, the motivation for this paper isn’t based so much on the relative frequency of non-PIE or data-in-code features, but on the research question as to whether support for these features must necessarily come at the cost of high overheads, or the possibility of arbitrary runtime failures.

### 1.2 Contributions

To the best of our knowledge, ours is the first binary instrumentation work to combine the performance benefits of rewrite-time code-pointer transformation with the flexibility and compatibility benefits provided by runtime address translation. The result is a system that can support binaries that bring in complexities such as embedded data and position-dependent code with modest overheads (∼5%) while incurring low overheads (∼2%) on well-behaved code.

Such a combination is enabled by a new multiplication-based encoding technique that can (a) distinguish transformed pointers from untransformed ones, and (b) detect, with high probability, alterations of a transformed pointer. This latter ability is crucial for achieving our goal of predictably failing in the face of pointer identification errors. Our encoding achieves this while only needing a single unused bit in addresses.

Ours is also the first work to formulate the problem of safe jump table transformation and to develop a solution. Our approach avoids the stringent constraints placed by in-place modification of jump tables, including the need for exact bounds, and a guarantee that some of the jump table entries aren’t used as data elsewhere in the binary. In addition, we ensure that any jump-table related code changes introduced by our instrumentation won’t lead to unintended changes in program behavior.

**Generality of approach.** Although our implementation targets 64-bit x86 systems running Linux, our techniques are broadly applicable to other platforms as well. In particular, problems of (code) pointer classification and data-in-code arise across different OSes and architectures. Similarly, our pointer encoding work relies on features common to modern platforms such as the availability of fast multiplication operations, a 64-bit address space, etc. Finally, jump table safety concerns transcend compilers and instruction sets, and hence our safe transformation techniques are also broadly applicable.

**Paper organization.** We begin with an overview of our disassembly technique in Sec. 2, followed by pointer classification in Sec. 3. We describe our pointer encoding and safe jump table

---

1Jump tables often result from the compilation of switch statements, but can also be in hand-coded assembly. They involve looking up a code offset by indexing into a data table and adding this offset to a base address. Such computed code pointers are particularly hard to reason about.

2CCFIR [58] makes similar assumptions for x86-32 binaries on Windows. Although C++ exception unwinding metadata is not stripped from the binary, exceptions and stack unwinding.

3Although C++ exception unwinding metadata is not stripped from the binary, exceptions and stack unwinding.
2.2 Fail-safe handling of false negatives

If the code corresponding to false negatives is reached during the execution of the original program, it is clear that the behavior of the instrumented program will diverge at that point since the corresponding instructions are not present in it. So the best that can be done is to detect this divergence and stop execution before the jump to the non-existent instruction. We follow the same approach as previous works (e.g., [54, 55, 62]) in this regard. They observe that any control transfers to unrecognized code must occur through indirect branches. Since indirect branches go through the address translation table and since this table won’t contain locations of unrecognized instructions (false negatives), this error will be stopped at the very first instruction that escapes recognized code.

Fail-safe handling prevents delayed and arbitrary failures, but it still leads to loss of functionality. So, these failures should be minimized as much as possible, as outlined below.

2.3 Our disassembly algorithm

Linear disassembly can achieve very low false negatives on most binaries, but has a high error rate on binaries that contain data within code. For this reason, we rely primarily on recursive disassembly [8, 22, 27, 44, 48] that is augmented with static analysis (described in Sec. 3) to discover code pointers.

The weakness of recursive disassembly is that it tends to have significant false negatives. Since false negatives lead to runtime failures, our disassembler is tuned for minimizing them at the cost of increased false positives. Specifically, we restart recursive disassembly at byte offsets where no code has been found by the initial phase of recursive disassembly. This step ensures that all bytes in the code section are recognized as part of some instruction, thus minimizing the chances of runtime failures. Similar to Multiverse [4], our disassembly algorithm allows for multiple interpretations of the same byte of code. The primary differences are that:

- Disassembly of an offset is attempted only if it is directly reachable from successfully disassembled code, or from code pointers discovered by the analyses in Sec. 3; and
- A conflict resolution algorithm is used to eliminate unlikely interpretations. This algorithm relies on several statistical properties of data and behavioral properties of code to identify candidates for elimination.

As a result, Multiverse disassembly increases the code size by about 5 times [4], while our approach experiences only a small increase. Due to space limitations, we omit the details of the disassembly algorithm here, but an interested reader can find them in [42]. Some of the key features of this algorithm are noted in Fig. 1.

3 Code Pointer Classification

The core of our approach consists of code pointer classification and code pointer transformation techniques that work together...
as illustrated in Fig. 2. We describe code pointer classification in this section, followed by transformation techniques for constant and computed code pointers in Sections 4 and 5.

### 3.1 Identifying code pointer constants

Examples of code pointer constants include function pointers and return addresses. They are characterized by the properties that their values are (a) represented by constants in the code or initialized data sections, and (b) used without any changes as the target of an indirect branch during the program run.

For position-independent binaries, pointer constants are marked for relocation. We flag such constants as tentative code pointers if their value falls within the binary’s code section.

For position-dependent binaries, we scan the data and code sections and flag any 64-bit and 32-bit constant falling within the code section as a tentative code pointer. Similar to BinCFI [62], these constants can occur at any byte boundary.

For both types of code, PC-relative constants loaded into registers are marked as tentative code pointers if they fall within the code region. All return addresses (i.e., locations following call instructions in the binary) are also marked.

**Definite vs possible code pointers.** In the next phase, each tentative pointer is classified either as a *definite* or *possible* code pointer. At instrumentation time, our method transforms definite code pointers so that they point to their intended target in the new code region. If this transformation is mistakenly applied to a non-pointer, the instrumented program’s behavior will be altered, leading to potential failures down the line. In particular, integer constants can have values that happen to fall in the range of the code addresses in a binary. Hence definite pointers can only include quantities known to be pointers:

- constant values marked for relocation.
- PC-relative addresses created using instructions of the form `lea offset (%rip),%rxx` where `rxx` represents a register.\(^4\)

\(^4\)Compilers generate this instruction when high-level code takes the address of a function.

(Note that the last two cases are applicable to position-dependent binaries as well.) Since all candidates for definite code pointer classification are known to be pointers, Fig. 2 does not need to consider the failure mode of definite pointers used as non-pointer data.

In the next step, two additional checks are applied to pointers that pass the above checks.

- the pointer value falls within the range of code addresses in the binary, and
- the code at the target matches the beginning of a function.

The first of these criteria has been used in many previous works, and the combination proposed in Uroboros [53]. One significant difference in our work is that we apply these criteria to quantities known to be pointers, whereas previous works have applied them (speculatively) to all constants. Another key difference is in the implementation of the second criterion. Specifically, we require the satisfaction of function interface properties [43]. Optionally, we add a strong version of function prolog pattern match [5, 8, 44, 48] that includes two or more push instructions, or one push followed by a decrement of the stack pointer [42]. The impact of this optional check is discussed in our evaluation.

Pointers passing the above checks are classified as definite code pointer constants, while all other tentative code pointers are classified as possible code pointers.

### 3.2 Identifying computed code pointers

This analysis is aimed at computing an overapproximation of all computed code pointers in a binary. We seek a superset because a missed code pointer will neither be transformed nor be included in the address translation table. As a result, if an instrumented program uses such a pointer, it will crash.

Similar to previous works, we assume that computed code

\(^5\)In our base implementation, return addresses in the original code are pushed on the stack. This approach is employed among Group I techniques to ensure compatibility with non-standard uses of return addresses.
pointers arise from the use of jump tables and that these tables only target intra-function locations. Hence our approach examines every indirect jump and analyzes the value flowing into this instruction. If this value’s computation matches one of the following patterns, the set of all its possible values is added to the set of computed code pointer targets.

- **Pattern 1**: Target = Base + Table[\(\text{index} + \text{stride}\)]
- **Pattern 2**: Target = Table[\(\text{index} + \text{stride}\)]
- **Pattern 3**: Target = Base + Table[\(\text{index} + \text{stride}\)]

These are more of semantic rather than syntactic patterns — in particular, Base, Table and stride don’t have to be constant literals in the binary but may be computed from other quantities. Our specific requirement is that their values be computable at instrumentation time using static analysis. In addition, Table must be located within code or read-only data sections of the binary.

To compute the list of all possible values of these jump table expressions, we need to compute the bounds of index. Because we undertake this possible analysis before disassembly is complete, we rely on a simpler approach that takes the lower bound as zero and the higher bound as the end of the binary, or the beginning of the next jump table. Values of Target for these index values are computed, and treated as possible code pointers if they fall within the code section of the binary.

Finally, there may be indirect jumps at which the static analysis described above fails. In that case, we identify the closest procedure surrounding this indirect jump and consider every instruction boundary within it as a valid target. This ensures that we have covered every possible indirect jump instruction in the binary and its possible values.

We examine the completeness of these three patterns in capturing jump table access in Table 1. We used the 64-bit binaries from Pang et al.’s dataset [36] for this evaluation. Note that these three patterns capture almost all of the jump targets.

In fact, the first two are sufficient to capture 99.8% of indirect jumps that access jump tables.

### 4 Safely Transforming Code Pointer Constants

The baseline for our approach is to leave code pointers unchanged, similar to Group I techniques. Runtime address translation is then used to dispatch indirect control flow transfers to their intended targets. However, this approach incurs high overheads. To avoid the translation overhead, Group II approaches [17, 57, 58] identify and “fix up” all code pointer constants at rewrite time, so that they reference the intended location in the instrumented code. This fixup involves adding an offset representing the distance between a code location in the original code, and the corresponding location in the instrumented code. Since instrumentation increases code length in an unpredictable manner, this “fix-up amount” differs for each location in the original code, which is why this kind of fixup can be performed only on constant code pointers.

A false negative in code pointer identification will mean that it wouldn’t have been fixed up, and hence won’t point to its intended target. Hence its use can cause a crash or undefined behavior of instrumented code. A false positive, on the other hand, may end up modifying an integer or data pointer value. Since this value differs between the original and instrumented versions, its use will cause a behavioral divergence between them, thus negating the functional correctness of the instrumented program.

Since static analysis techniques cannot guarantee 100% accuracy in code pointer identification, rewrite-time fix-up is applicable only to binaries containing relocation information. In particular, position-independent binaries on x86/Linux identify all pointer constants appearing within them. To identify the subset of these that are code pointers, two additional assumptions are made by Group II techniques:

- **There is no data embedded within code.** This enables any pointer constant whose value falls within the binary’s code section to be marked a code pointer.
- **Code pointer constants are never involved in computations.**

— since the fixup amount depends on the pointer value, any arithmetic on the pointer will invalidate it.

We now present a new technique that realizes most of the performance benefits of Group II approaches in the handling of code pointer constants, while relaxing these stringent assumptions.

#### 4.1 Approach Overview

The goals of our approach are to:

- **Tolerate all false negatives in definite code pointer classification**, including:
  - unidentified code pointers, and more importantly,
  - code pointers derived from data or other code pointers.

  This is achieved using the address translation technique described in Sec. 4.3.

- **Provide fail-crash behavior on false positives** where a value identified as a code pointer constant is used to:
  - dereference data, possibly after pointer arithmetic, or
  - compute a different code pointer.

<table>
<thead>
<tr>
<th></th>
<th>Number of indirect jumps using a jump table</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Groundtruth</td>
<td>60165</td>
<td>100%</td>
</tr>
<tr>
<td>Detected</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pattern 1</td>
<td>16102</td>
<td>26.76%</td>
</tr>
<tr>
<td>Pattern 2</td>
<td>43957</td>
<td>73.06%</td>
</tr>
<tr>
<td>Pattern 3</td>
<td>4</td>
<td>0.01%</td>
</tr>
<tr>
<td>Missed by the 3 patterns</td>
<td>102</td>
<td>0.17%</td>
</tr>
</tbody>
</table>

Table 1: Completeness of patterns to identify jump-table related indirect jumps.

---

6This is an assumption since there isn’t an easy way to verify it with static analysis: pointer constants may be stored in global memory by one function, and used at an arbitrary time later by another function. It is very difficult for static analysis to reason about dataflows through global memory in binaries.
This goal is achieved using a pointer encoding technique (Sec. 4.2) that can distinguish fixed-up pointers that haven’t been subsequently altered from all other types of pointers.

The following instrumentation, added at indirect control transfers, determines whether address translation or pointer decoding is to be used:

```plaintext
if encoded(target) then
    target ← decode(target)
else
    target ← address_translate(target)
end if
Transfer control to target
```

### 4.2 Code Pointer Encoding

Pointer encoding has two major goals:

1. determine if an indirect target has already been fixed up;
2. detect modifications of fixed-up pointers.

The first goal is relatively easy to achieve on 64-bit platforms, where the leading 16 bits of the address are typically unusable. In particular, we can set the most significant bit (MSB) of a fixed-up address to be “1,” while untransformed addresses will have a leading zero-bit. Note that transformed addresses point to unmapped memory (since the MSB is always zero for valid user-land addresses), so if they are used for data access, it will lead to a memory fault. Thus, the use of encoded pointers to access data will meet the design goal of a fail-crash.

However, the second goal is more difficult. One possibility is to compute a 15-bit checksum and store it within the leading 16-bits. This checksum can be recomputed in the indirect branch instrumentation and checked. A mismatch indicates the pointer has been modified. But this checksum approach has a few drawbacks:

- Checksum computation takes time, reducing performance.
- A 15-bit checksum means a non-negligible 1 in 32K chance of failing to catch a modification, and
- Some applications may already be using the leading 16-bits since they are known to be unused on the platform. This is a well-known technique that goes by several names such as pointer-packing, boxing, and pointer tagging. Many functional programming languages as well as JavaScript engines use this kind of packing, e.g., V8’s NaN boxing.

We, therefore, present a simple and elegant alternative that can provide probabilistic detection based on the fact that only a tiny fraction of the available 48-bit address is actually used by any program. It does not need any additional bits to hold a checksum and can simply work with how many ever bits are actually used for addresses.

Our approach is configurable and takes as input the number of usable bits \( n \) for pointers. We will set the most significant of these \( n \) bits to be a “1” for encoded pointers and a “0” for unencoded ones. (We continue to assume that this MSB cannot be a “1” for valid addresses.)

Our encoding relies on a single 64-bit multiplication. Contemporary Intel and AMD’s processors have very fast multiplication units that can deliver a throughput of one multiplication per clock cycle, with a latency of just 3 to 4 cycles [23]. Moreover, a MULX instruction is available that does not affect CPU flags, and hence can be used without the added overhead of saving/restoring flags. We leverage these facts together with the following observation:

**Observation 1 (Multiplicative Inverse Modulo \( 2^n \))** Every odd number \( A < 2^n \) has a unique multiplicative inverse modulo \( 2^n \), i.e., there is a \( B < 2^n \) such that \( AB \equiv 1 \mod 2^n \).

**Proof:** This is a known result, but since the proof is simple and is also the basis for Observation 3, we present it here. Consider the following list of numbers, where all multiplications are performed modulo \( 2^n \):

\[
A, 3A, 5A, ..., (2^n - 1)A
\]

We now show that all these products are distinct. Otherwise, there must be two odd numbers \( X, Y < 2^n \) such that

\[
AX \equiv AY \mod 2^n; \text{ in other words } A(X - Y) \equiv 0 \mod 2^n
\]

Since \( A, X, \) and \( Y \) are all odd, they have no common factors with \( 2^n \). So the only way for \( A(X - Y) \equiv 0 \mod 2^n \) to hold is if \( X \equiv Y \mod 2^n \). Finally, since \( X, Y < 2^n \), this means \( X = Y \).

This means that all the \( 2^{n-1} \) products listed above are distinct. Moreover, they are all odd because the product of two odd numbers is always odd. Since there are exactly \( 2^{n-1} \) odd numbers less than \( 2^n \), this means that the above list is a permutation of \( 1, 3, 5, ..., 2^n - 1 \). Thus, there must be some odd number \( B \) such that \( AB \equiv 1 \mod 2^n \). ■

Extended Euclidean Algorithm [56] can be used to calculate this multiplicative inverse efficiently. In any case, this computation is a one-time cost for our algorithm. Now we are ready to describe our encoding technique:

**procedure init**

1. Generate an odd random number \( A \) uniformly distributed over \( 1...2^{n-1} \).
2. Compute \( B \) such that \( AB \equiv 1 \mod 2^{n-1} \).

**function enc(x) = (Ax \mod 2^{n-1}) \cdot 2^{-1}**

**function dec(x) = B \cdot (x \& (2^{n-1} - 1)) \mod 2^{n-1}**

Note that \( x \mod 2^k \) can be efficiently computed using a single bitwise-and operation, specifically, \( x \& (2^k - 1) \).

**Observation 2 (Encoding Scheme Properties)**

- \( \forall x \ dec(enc(x)) = x \).
- \( \forall y \ if \ y = O \cdot 2^i \ where \ O \ is \ odd, \ then \ dec(y) \ is \ uniformly \ random \ over \ \{ 2^i, 3 \cdot 2^i, 5 \cdot 2^i, ..., (2^{n-1}-1) \cdot 2^i \} \).

The first point follows from the definition above:

\[
dec(enc(x)) \equiv BAx \equiv (BA)x \equiv x \mod 2^{n-1}
\]
The condition \( y = O \cdot 2^{r} \) in the second point corresponds to \( y \) being an address that has \( r \)-bits of alignment. When such a \( y \) is decoded, the product operation will still leave the last \( r \) bits as zero. Using the same logic as in the proof of Observation 1, it can be shown that there are exactly \( 2^{r} \) distinct values of \( B \) that will produce each of the values in the set \( \{ 2^{r}, 3 \cdot 2^{r}, 5 \cdot 2^{r}, ..., (2^{n-1} - r - 1) \cdot 2^{r} \} \).\(^7\) In other words, the product \( B \cdot y \) is uniformly random over this set. Based on this we can now show that the encoding scheme detects alterations to an encoded pointer.

**Observation 3 (Correctness of Encoding Scheme)** For a memory address \( p \), let valid\((p)\) denote that \( p \) falls within an executable memory page. Also, let \( S \) denote the total number of bytes mapped for execution within the address space of a process. Then:

- If \( y = enc(x) \) and valid\((x)\) then \( Pr[valid(dec(y))] = 1 \).
- Otherwise, \( Pr[valid(dec(y))] = \frac{S}{2^{n-1}} \).

**Proof:** The first part is just a restatement of the first point from the last observation. To establish that second point, note that if \( y = O \cdot 2^{r} \), then, from the second point of the last observation, \( dec(y) \) can range over \( 2^{n-r-1} \) values. Assuming that (a) the address space mapped for execution is also uniformly random, and (b) considering the fact that among \( S \) such addresses there will be \( \frac{S}{2^{r}} \) that will have \( r \) bits of alignment, the likelihood that \( B \cdot y \) is one of these valid addresses is:

\[
\frac{S}{2^{n-1}} = \frac{S}{2^{n-1}}
\]

This observation means the following: if the program attempts an indirect control transfer using a pointer \( y \) that was altered after encoding, then it will lead to a fail-crash (through a memory fault) with probability \( \frac{S}{2^{n-1}} \). Considering that the largest binaries are typically under 128MB = \( 2^{27} \)B, and that \( n \) is at least 48, there is only one in a million chance (or more precisely, 1 in \( 2^{20} \)) of failing to detect any arithmetic may have been performed on an encoded pointer. If all 64 bits of the pointer are usable, this decreases further to 1 in 64 billion. For a typical binary of size <8MB, it is 1 in a trillion.

### 4.3 Address Translation (AT)

As noted before, our approach relies on address translation to tolerate false negatives in code pointer classification. For modularity, address translation uses a two-level structure: a global translation table (GTT) and a per-module local translation table (LTT). The LTT is constructed at instrumentation time. In contrast, the GTT is constructed at runtime, with new entries added as and when a binary is loaded.

Due to ASLR, the absolute values of all code pointers can only be determined at runtime. So, we store offsets — the relative distance between a code pointer and the base of the code section — in the LTT, since the offsets are fixed for a given binary. For each possible code pointer \( P \) identified by our analysis in Sec. 3, we add the pair \((O_x, O_y)\) to the LTT, where \( O_x \) is the offset of \( P \) in the original code, and \( O_y \) is the corresponding offset in the new code. This hash table uses open-addressing so as to avoid linked lists that can be cache-unfriendly.

At runtime, when a code pointer \( P \) is used in an indirect control transfer by a binary, it is not possible to predict which other binary \( P \) came from. So, the lookup of \( P \) always goes to the GTT. As described in Sec. A, instrumented binaries use a customized loader that populates the GTT whenever a binary is loaded. Specifically, an entry is created in the GTT for each code page of the binary. This entry maps the page address (leading 52 bits of the address) to the LTT of the binary. The GTT is also structured as a hash table with open addressing.

#### 4.3.1 Optimization: Avoiding address translation for returns

Since returns are very frequent, they exacerbate the overhead of address translation. Using native calls and returns will relieve this overhead. However, this leads to incompatibilities with non-standard uses of return addresses, e.g., C++ exception handling uses return addresses on the stack to determine the landing pad\(^8\) for an exception. Such non-standard uses will break if the original return addresses are replaced with the corresponding new locations in the instrumented code.

Other non-standard use of return addresses are (a) as targets of an indirect call or jump, or (b) to compute data locations. The second use is only observed on 32-bit x86 processors due to their lack of support for PC-relative addressing. Since (a) is already handled in our approach by including all possible return addresses in the LTT, the only compatibility issue raised is due to C++ exceptions. We, therefore, implemented an optimization wherein we update all the exception handling metadata to reflect the location of the new code. To support indirect jumps within the exception handling module, we add these new landing pad pointers to LTT as well. Finally, to support the uses of return addresses as indirect call/jump targets, we need to add the new return addresses (rather than the original ones) to the LTT. With this optimization place, we can avoid instrumentation for (direct) calls and (all) returns, leading to substantial reductions in overhead.

### 5 Safe Jump Table Instrumentation

Most Group I approaches \([4, 7, 32, 62]\) do not try to analyze or predict the value of a computed code pointer, but simply route its actual runtime value through the address translation table. Unfortunately, this approach exacts a high performance cost when jump tables are used in inner loops. This prompted efforts to optimize away this address translation step when possible \([62]\). Group II approaches essentially apply this optimization in all cases since there is no address translation to fall back on. This requires updating the jump table contents

\(^7\)Basically, the leading \( r \) bits of \( B \) don’t affect the lower \( r \)-bits of \( B \cdot O \cdot 2^{r} \).

\(^8\)This is essentially the catch block designated to handle this exception.
so that it now stores offsets in the \textit{instrumented} code rather than the \textit{original} code. This raises several major challenges for static analysis as well as instrumentation correctness:

1. Identifying the exact bounds (base and size) of a jump table.
2. Identifying the exact set of index values and target locations used in a jump table.
3. Jump table is essentially some data used by the program under instrumentation. Changing its content can thus amount to data corruption, the result of which could be a crash or an arbitrary failure.

The first two challenges — which relate to the completeness of jump table recovery — have been recognized and discussed in previous works. According to Pang et. al \cite{36}, Angr \cite{48}, Dyninst \cite{5}, and Ghidra \cite{44} respectively miss about 24\%, 1\%, and 8\% of the jump table targets. Egalito \cite{57} reports 1\% false negatives, but the table bounds were missed at a much higher 6\% rate. There is generally no recourse for these false negatives — if the instrumented program uses a missed target, it will cause (unpredictable) failure. Thus, the first goal we address in this paper is the development of a technique in Sec. 3.2 that is geared to identify all possible targets.

The third challenge — on instrumentation soundness — has not received much attention in previous work. If jump table bounds are not correctly identified, then these techniques can corrupt adjacent data. If the program uses some of a jump table’s content as data, the instrumentation would break them. Previous works \cite{17, 41, 57, 62} are essentially best-effort solutions in this regard, with correctness contingent on these factors. In contrast, we formulate the criteria for safe jump table transformation in Sec 5.1 and present a method to realize it.

5.1 Tolerant jump table transformation

To modify the jump table safely, we need to ensure that its contents are used solely for jump table access. If there are other uses of this data, possibly from functions other than the one using the jump table, then jump table modifications will affect this use and most likely result in misbehavior of the instrumented program. One possible approach is to develop a static analysis to rule out such uses, but this does not seem promising. In particular, data accesses frequently involve complex pointer arithmetic that is very hard to reason about, especially in binaries. Hence, we develop a different approach that leaves the original jump table intact, thus preserving the behavior of any code that uses it as data.

Our approach is to make a copy of the jump table and modify the code that used the original jump table to use this copy. For instance, the code accessing a jump table using the expression $\text{Base} + \text{Table}[\text{index}]$ will be transformed into one that accesses $\text{NewBase} + \text{NewTable}[\text{index}]$, where $\text{NewTable}$ is the new location of the jump table. Contents of this relocated jump table will be changed so that the $\text{NewBase} + \text{NewTable}[\text{index}]$ will point to the new location of the code accessed by $\text{Base} + \text{Table}[\text{index}]$ for each value of $\text{index}$. Note also that in this transformation, exact bounds information is not essential for soundness. For instance, if we overestimate the size of the table, we end up using more storage, but there is no correctness impact since the entries beyond the correct bound won’t be accessed by the program.

There are instances where the same jump table is used by more than one indirect jump. To ensure safety, we analyze each indirect jump separately and make a separate copy of the jump table for each of them. Although this may seem wasteful, multiple uses are infrequent, so this does not have a significant impact on memory overhead.

5.2 Safe transformability analysis

The above approach makes changes to the code for accessing/using the jump table. We need to ensure that these changes don’t have side effects that can change program behavior. To do this, note that the changes described above only involve substituting one set of constant operands (specifically, $\text{Base}$ and $\text{Table}$) with another set. Thus, all possible effects of this change can be identified by performing a dependence analysis starting from these instructions.

Specifically, we use a static taint analysis that marks these new operands as tainted. After propagating this taint through the current function, the following checks are made:

- \textit{Deref}: Tainted value should not be used to dereference memory except (a) the intended memory read, namely, the instruction that originally read $\text{Table}[\text{index}]$, and (b) the indirect jump used by this jump table.
- \textit{StoreMem}: Tainted value should never be stored in memory.
- \textit{CallArgs/RetVal}: Tainted value should never escape outside the current function, e.g., it should not be passed as a parameter to a callee, or be returned to the caller of the current function.

If all these checks are satisfied, then it is clear that the new jump table contents are used solely in computing the indirect jump target, and that it does not affect anything else. In other words, it is safe to perform the jump table transformation. If any of the checks are violated, then we conservatively leave the jump table as-is, and instrument the indirect jump to use address translation.

6 Evaluation

Our experiments evaluate \textsc{SAFER} along the following axes:

- functionality, including correct instrumentation of complex binaries, handling data in code, supporting non-PIEs, and failing safely when assumptions are violated (Sec. 6.2);
- runtime overhead, effectiveness of \textsc{SAFER} optimizations, and the impact of compiler optimization levels (Sec. 6.3);
- memory overhead (Sec 6.4); and
- support for security-relevant instrumentation (Sec. B).
6.1 Experimental setup

Datasets.

- **Dataset 1 (473.5 MB):** 15 pre-built real-world programs (Table 2) and the corresponding 572 shared libraries used by these programs.

- **Dataset 2 (47.6 MB):** SPEC 2006 benchmark has been used in most previous works and provides a basis for direct performance comparisons. It was compiled with gcc, g++, and gfortran at the default optimization level (-O2).

- **Dataset 3 (587.2 MB):** SPEC 2017 benchmark, a better stress-test for SAFER due to its much larger size.

- **Dataset 4 (30 MB):** Custom-compiled coreutils, used to systematically evaluate SAFER’s ability to instrument binaries with data embedded in code and its fail-crash design.

Instrumentation settings. Similar to previous works in static/dynamic binary instrumentation, most of the evaluation is performed with null instrumentation. This means that no application-specific instrumentation is added, but all of the components of SAFER are exercised, including disassembly, pointer analysis, pointer encoding and decoding, address translation, and safe jump table transformation. The only exception is Sec. B which adds CFI and shadow stack instrumentation.

Evaluation platform. All experiments were carried out on a desktop running a 12th generation Intel Core i7 processor with 16 GB main memory and 500 GB SSD, running Ubuntu 20.04.

6.2 Functionality evaluation

Our functionality evaluation aims to answer these questions:

- Does rewriting preserve functionality? (Sec. 6.2.1)
- Can it correctly rewrite binaries that cannot be handled by Group II approaches? This includes non-PIE binaries and binaries containing data within code. (See Sec. 6.2.2.)
- In cases where SAFER’s assumptions are violated, does it fail safely? (See Sec. 6.2.2.)

6.2.1 Position-independent executables and libraries

We tested SAFER functionality using SPEC 2006 and 2017 benchmark suites, coreutils, and several real-world applications (Table 2). We transformed the executables and all the libraries used by these programs. The total size of the code transformed, including the SPEC binaries, was around 1.1GB. Among SPEC benchmarks, `uninstrumented` versions of wrf and games in SPEC 2006 and cam4 in SPEC 2017 failed on our experimental platform, so we omitted them from our evaluation.

The SPEC suite and coreutils come with their own set of functionality tests. For the real-world applications, we show the test cases used in Table 2.

Note that even with all optimizations enabled, SAFER preserves compatibility with C++ exception handling, stack tracing and `longjmp`’s. SPEC 2006 omnetpp and povray use exception handling at runtime and their instrumented version executed without any failure. Perlbench benchmark uses `longjmp` and it executes successfully post instrumentation.

6.2.2 Binaries with data-in-code and non-PIEs

Customized coreutils with embedded data To further evaluate the ability of SAFER to handle binaries that challenge disassembly and code pointer identification techniques, we compiled the coreutils package with data embedded in the middle. We used a custom linker script for this purpose. Instead of creating a separate section of the program’s read-only data, this linker script embeds it in the code section (`.text`). Note that we did not add any additional data manually. Instead, the linker script scatters the program’s read-only data from different object files (`.o`) across the code section.

We first used this data-in-code dataset to evaluate if SAFER achieves its fail-crash objective when its disassembly and/or code pointer identification fails. Table 3 shows the results of this experiment. Data-in-code causes our disassembly to misclassify some instructions, but as described in Sec. 2, our instrumentation algorithm can tolerate these errors fully. So, we did not anticipate any instrumentation time or runtime failures related to incorrect disassembly. We confirmed this by instrumenting the binaries in full address translation mode, where none of the pointers are fixed up and the program relied on

<table>
<thead>
<tr>
<th>Program</th>
<th># of libs</th>
<th>Size (MB)</th>
<th>Data-in-code libs</th>
<th>Test case</th>
</tr>
</thead>
<tbody>
<tr>
<td>gimp</td>
<td>279</td>
<td>190</td>
<td>gcrypt gnuTs avcodec</td>
<td>Create a drawing and save file</td>
</tr>
<tr>
<td>ffmpeg</td>
<td>187</td>
<td>175</td>
<td>gcrypt gnuTs</td>
<td>Convert a MP4 video of size &gt; 50MB to MKV</td>
</tr>
<tr>
<td>clang (non-PIE)</td>
<td>15</td>
<td>122</td>
<td></td>
<td>Compile 50+MB program</td>
</tr>
<tr>
<td>evince</td>
<td>148</td>
<td>115</td>
<td>gcrypt gnuTs</td>
<td>Open a PDF file, view pages</td>
</tr>
<tr>
<td>apt</td>
<td>19</td>
<td>113</td>
<td>gencrypt</td>
<td>Run “apt-get upgrade”</td>
</tr>
<tr>
<td>gedit</td>
<td>117</td>
<td>81</td>
<td>gencrypt</td>
<td>Open a text file, edit and save</td>
</tr>
<tr>
<td>vim</td>
<td>20</td>
<td>13</td>
<td></td>
<td>Open a file, edit and save</td>
</tr>
<tr>
<td>Python (non-PIE)</td>
<td>8</td>
<td>9</td>
<td></td>
<td>Run Pystone benchmark</td>
</tr>
<tr>
<td>pdflatex</td>
<td>8</td>
<td>8</td>
<td></td>
<td>Compile .tex files with total size of 100KB</td>
</tr>
<tr>
<td>scp</td>
<td>5</td>
<td>5</td>
<td>crypto</td>
<td>Copy files of size &gt; 100MB</td>
</tr>
<tr>
<td>find</td>
<td>7</td>
<td>4.5</td>
<td></td>
<td>Find files in /usr/bin by name</td>
</tr>
<tr>
<td>tmux</td>
<td>8</td>
<td>3.5</td>
<td></td>
<td>Open a tmux session, issue commands to terminal, quit</td>
</tr>
<tr>
<td>tar</td>
<td>7</td>
<td>3</td>
<td></td>
<td>Compress a 2+GB directory</td>
</tr>
<tr>
<td>enscript</td>
<td>4</td>
<td>3</td>
<td></td>
<td>Convert text file of size &gt; 100MB to PDF</td>
</tr>
<tr>
<td>gcc (non-PIE)</td>
<td>2</td>
<td>3</td>
<td></td>
<td>Compile 50+MB program</td>
</tr>
<tr>
<td>make</td>
<td>3</td>
<td>2</td>
<td></td>
<td>Compile a program with more than 100 source files</td>
</tr>
</tbody>
</table>

| Total    | 572       | 473.5     |                   |           |

Table 2: Functionality testing on real-world applications. Size indicates the combined size of a binary and all the libraries used by it. Library names are shown without the `lib` prefix and `.so` suffix.
This shows that secure instrumentation of real-world applica-

validation (i.e., function prolog-based validation disabled).

encounter crashes and need to initiate reinstrumentation.

apply this fall-back mode of instrumentation for such binaries.

have a high likelihood of containing data, and proactively

information from the disassembly phase to flag binaries that

full address translation. All the related programs executed

function prolog-based validation and (ii) the second time with

instrumenting just this library twice: (i) the first time with

of S

approach. We could isolate the problem to this module because

libgcrypt.so

be instrumented successfully with ABI-based code pointer

libraries containing data embedded in code, two of them could

tions requires the ability to support data in code. Of the three

Table 3: Coreutils with data embedded in code.

<table>
<thead>
<tr>
<th>Library</th>
<th>Rewrite success</th>
<th>Execution success</th>
<th>Safe Failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>runtime AT</td>
<td>105/105</td>
<td>105/105</td>
<td>NA</td>
</tr>
<tr>
<td>Instruction boundary</td>
<td>105/105</td>
<td>43/105</td>
<td>62</td>
</tr>
<tr>
<td>ABI validation</td>
<td>105/105</td>
<td>74/105</td>
<td>31</td>
</tr>
<tr>
<td>Function prolog matching</td>
<td>105/105</td>
<td>105/105</td>
<td>NA</td>
</tr>
</tbody>
</table>

runtime address translation. The first row (runtime AT) in Ta-

ble 3 summarizes the results. As expected, all the tests passed.

Next, we tested if our objective of predictable failure is met if
code pointers are misclassified. We tried three techniques:

• Instruction boundary based: This technique designates any

pointer that references an instruction boundary for pointer

encoding. This results in multiple misclassifications, and

as a result, about 60% of the test cases failed. We verified

in each case that the failure was due to the dereferencing

data using an encoded pointer. This is the failure mode
designed into the system, and our experiment shows that it

is achieved. We investigated the error further in gdb, and

found that most errors were due to a string argument pointer

that was passed to a print routine in glibc.

• ABI-based: In this mode, we designate a pointer as a code

pointer if satisfies the ABI-designated properties that all

functions must respect. There can be small data fragments

that disassemble into code that doesn’t violate the ABI, and

hence this test alone is not a reliable indicator of function

pointers. As a result, almost 30% of the test cases fail. As in

the previous case, we verified that the failures happened in

the predicted manner.

• ABI and function prolog matching: With this setting, SAFER

is able to correctly classify code pointers despite the fact

that these binaries contained data within code. As a result,

all tests were passed.

Real-world applications using libraries with data embedded

in code. Observe that nearly 40% of the applications in

Table 2 use a library that contains data embedded within code.

This shows that secure instrumentation of real-world applica-
tions requires the ability to support data in code. Of the three

libraries containing data embedded in code, two of them could

be instrumented successfully with ABI-based code pointer

validation (i.e., function prolog-based validation disabled). But

libgcrypt.so had a data pointer misclassified using this

approach. We could isolate the problem to this module because

of SAFER’s safe-failure (fail-crash) design. We retested by

instrumenting just this library twice: (i) the first time with

function prolog-based validation and (ii) the second time with

full address translation. All the related programs executed

successfully both times. In future work, we plan to leverage

information from the disassembly phase to flag binaries that

have a high likelihood of containing data, and proactively

apply this fall-back mode of instrumentation for such binaries.

Such an approach can minimize the instances where users

counter crashes and need to initiate reinstrumentation.

Position-dependent (non-PIE) binaries. We first recom-

piled SPEC 2006 into position-dependent code (non-PIE). Note

that Group II approaches such as Egalito [57], RETROWI-

te [17] and SBR [41] cannot handle such binaries.

We verified that SAFER can transform these non-PIEs and

they passed all the tests. Among real-world applications, gcc,
clang and python are non-PIEs.

6.3 Runtime Overhead

In this section, we aim to answer the following questions:

• How do SAFER’s overheads compare with previous works

in Group I and Group II? (Secs. 6.3.1, 6.3.2.)

• Is SAFER efficient across different optimization levels used

to compile a binary? (Sec. 6.3.3.)

• How effective are its design features and optimization tech-
niques in reducing runtime overhead? (Sec. 6.3.4.)

• What is its performance on non-PIEs? (Sec. 6.3.5.)

Our answers to these questions are based on the results shown

in Figs. 3, 4 and Table 4. Fig. 3 shows the performance on

SPEC 2006 binaries compiled with gcc at O2 optimization

level and PIE format. Fig. 4 shows non-PIE versions of the

same programs compiled at the same optimization levels.

Table 4 includes programs from SPEC 2017 benchmarks. For

all of the experiments, we instrumented the whole program,
including all the shared libraries.

6.3.1 Comparison with Group II Techniques

Group II techniques [17,41,57] prioritize performance over

compatibility and graceful error-handling, and hence are

applicable only to position-independent binaries without

embedded data. In addition, if jump tables don’t match

one of the patterns supported by the tool, it can lead to

instrumentation or runtime failures. With these assumptions,

they are able to achieve an overhead that is close to zero.

SAFER does not make these assumptions yet comes close in

terms of performance, achieving an overhead of 1.91%.

6.3.2 Comparison with Group I Techniques

Group I techniques require fewer assumptions but incur

significantly higher overheads than Group II techniques.

BinCFI [59] without any additional optimizations incurs

34.33% overhead. The corresponding number reported by

MULTIVERSE authors [4] is 60%. The comparable number for

SAFER is 30.8%, obtained in “full AT” mode. (This mode

uses original code addresses everywhere, including return

addresses, thus requiring address translation for each return

and all indirect calls and jumps.)

Both BinCFI and MULTIVERSE present a few optimizations

similar to ours, e.g., return address (RA) optimization.

However, the configurations are not comparable. For instance,

Multiverse reports RA optimization when the libraries are not

instrumented. Moreover, performance numbers are reported

only for 12 of the 29 SPEC 2006 benchmarks. Still, the best
numbers reported by these works are 8.54% for BinCFI and 8.29% for Multiverse. In comparison, SAFER’s overhead with all optimizations enabled is 1.91%. This is achieved because (a) our pointer encoding technique can take advantage of position-independent binaries to achieve faster performance while retaining the ability to use address translation for (the few) libraries that may require it, and (b) our jump table transformations guarantee safety for all binaries.

6.3.3 Effect of compiler optimization levels

In this section, we evaluate if our approach is effective consistently across different optimization levels used to compile the instrumented binaries. We consider both its runtime overhead and correctness here. We used SPEC 2017 benchmark suite, as it is much larger than SPEC 2006 and hence can provide a more robust test for instrumentation systems such as ours. The benchmarks were compiled into PIEs, the default on our experimental platform.

Table 4 summarizes our results. SAFER correctly instruments all binaries for all optimization levels, passing all the tests included in the benchmark suite. Recall that the uninstrumented version of cam4 fails across all optimization levels and hence is not included in the table. All other SPEC 2017 benchmarks are shown in the table. Uninstrumented versions of three other applications wrf, bwaves and roms fail with Ofast optimization, resulting in three “not applicable” (NA) entries in the table.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>O0</th>
<th>O1</th>
<th>O2</th>
<th>O3</th>
<th>Ofast</th>
<th>Os</th>
</tr>
</thead>
<tbody>
<tr>
<td>perlbench</td>
<td>3.61</td>
<td>25.31</td>
<td>7.05</td>
<td>8</td>
<td>5.92</td>
<td>17.93</td>
</tr>
<tr>
<td>gcc</td>
<td>10.97</td>
<td>2.54</td>
<td>3.4</td>
<td>2.27</td>
<td>2.64</td>
<td>1.49</td>
</tr>
<tr>
<td>mcf</td>
<td>5.31</td>
<td>6.21</td>
<td>6.25</td>
<td>6.46</td>
<td>7.95</td>
<td>6.34</td>
</tr>
<tr>
<td>omnetpp</td>
<td>4.58</td>
<td>11.54</td>
<td>12.05</td>
<td>10.89</td>
<td>11.6</td>
<td>12.23</td>
</tr>
<tr>
<td>xalancbmk</td>
<td>1.47</td>
<td>3.64</td>
<td>5.59</td>
<td>5</td>
<td>5.07</td>
<td>5.23</td>
</tr>
<tr>
<td>x264</td>
<td>-2.05</td>
<td>-0.42</td>
<td>-1.65</td>
<td>-1.83</td>
<td>-2.69</td>
<td>-0.41</td>
</tr>
<tr>
<td>deepjeng</td>
<td>0.48</td>
<td>-0.4</td>
<td>0.34</td>
<td>0.43</td>
<td>1.3</td>
<td>-0.77</td>
</tr>
<tr>
<td>leela</td>
<td>-0.1</td>
<td>-0.61</td>
<td>0.63</td>
<td>0.68</td>
<td>0.68</td>
<td>-0.59</td>
</tr>
<tr>
<td>exchange2</td>
<td>0.82</td>
<td>3.13</td>
<td>2.7</td>
<td>0</td>
<td>2.8</td>
<td>1.37</td>
</tr>
<tr>
<td>xz</td>
<td>0.19</td>
<td>0.14</td>
<td>0.44</td>
<td>0.44</td>
<td>0.44</td>
<td>0.28</td>
</tr>
<tr>
<td>bwaves</td>
<td>0</td>
<td>0.13</td>
<td>0</td>
<td>0.13</td>
<td>NA</td>
<td>0</td>
</tr>
<tr>
<td>cactuBSSN</td>
<td>0.71</td>
<td>0.64</td>
<td>-1.27</td>
<td>0.63</td>
<td>0.63</td>
<td>0.6</td>
</tr>
<tr>
<td>ibm</td>
<td>-0.38</td>
<td>0.13</td>
<td>-0.13</td>
<td>-0.38</td>
<td>0</td>
<td>0.13</td>
</tr>
<tr>
<td>wrf</td>
<td>0.12</td>
<td>0.88</td>
<td>0.84</td>
<td>0.34</td>
<td>NA</td>
<td>0.46</td>
</tr>
<tr>
<td>pop2</td>
<td>-0.41</td>
<td>-1.28</td>
<td>1.09</td>
<td>-0.22</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>imagick</td>
<td>0.09</td>
<td>0.21</td>
<td>0</td>
<td>-0.54</td>
<td>-0.34</td>
<td>0</td>
</tr>
<tr>
<td>nab</td>
<td>1.93</td>
<td>7.19</td>
<td>2.44</td>
<td>1.22</td>
<td>2.07</td>
<td>4.44</td>
</tr>
<tr>
<td>fotomik3d</td>
<td>0.59</td>
<td>0.96</td>
<td>0.66</td>
<td>0.66</td>
<td>0.67</td>
<td>0.99</td>
</tr>
<tr>
<td>roms</td>
<td>0.53</td>
<td>1.38</td>
<td>0.95</td>
<td>0.96</td>
<td>NA</td>
<td>0.46</td>
</tr>
<tr>
<td>Average</td>
<td>1.5</td>
<td>3.22</td>
<td>2.2</td>
<td>1.85</td>
<td>2.42</td>
<td>2.64</td>
</tr>
</tbody>
</table>

Table 4: Performance overheads for SPEC 2017 benchmarks. All numbers are in %. Uninstrumented or base version of benchmarks with NA failed on our experimental platform.

The average overhead on SPEC 2017 across all optimization levels is 2.3%. With O3 optimization (default for SPEC 2017), the overhead is 1.85%.
6.3.4 Effectiveness of SAFER optimizations

In this section, we evaluate the effectiveness of the key optimizations introduced in this paper. Our evaluation results are shown in Fig. 3. They use SPEC 2006 because its use cases comparison with previous works that have all relied on it.

The base version of SAFER uses full address translation (“full AT”), capturing the behavior of Group I systems. Note that full transparency is preserved in this case, as all the (pointer) values stored in memory will correspond to that of the original code. This ensures the preservation of application behavior. In this mode, all indirect control transfers, including all returns, go through address translation. The average overhead is 30.8% for SPEC 2006.

Code Pointer Encoding. This is the primary optimization introduced in the paper, and it enables efficient treatment of definite code pointers to coexist with the transparent treatment needed for possible pointers to work correctly. Moreover, this technique underpins the safety of our system, ensuring that errors in pointer classification will crash predictably before they can cause more serious failures.

To evaluate the performance benefits of pointer encoding, we compare the overhead between “full AT” and “full encoding,” which uses pointer encoding (instead of AT) selectively for definite pointers. Note that return addresses pushed on the stack by call instructions are definite code pointers, so they are encoded as well.\(^9\)

The use of encoding cuts down the average overhead by about 3×, bringing it down from 30.8% to 10.6%.

Return Address (RA) Optimization. Returns far outnumber indirect calls and jumps in most programs and hence improving their treatment can have an outsize impact on total overhead. Many previous Group I techniques have hence proposed an optimization to store the current rather than the original return address on the stack. Unfortunately, this change often breaks complex programs that make non-standard uses of return address, including as a call target, in C++ exception handling, etc. The key distinction in our approach, described in Sec. 4.3.1, is that we preserve these non-standard uses, and hence do not pay a price in terms of compatibility.

The RA optimization reduces the overhead by another 2×, bringing it down to 4.5%.

Safe Jump Table Transformation. Finally, the “JT Optimization” bars in Fig. 3 reveal the impact of our jump table optimization described in Sec. 5. Note that the overhead reduces by approximately another 2× to ≈ 1.91%. Further examination shows that this overhead reduction is modest for most programs, but has a major impact on perlbench, reducing its overhead from 53% to 12%.

Fig. 5 shows that about 55% of the jump tables are classified as “safely transformable” by our static analysis. The remaining jump tables are deemed unsafe for the reasons listed in Sec. 5.2. As indicated in red color, the biggest culprit is CallArgs, where a tainted value (i.e., a value that is modified by the new jump table computations) is possibly passed as an argument to a called function. Digging into this further, we found the following reason: our current analysis treats every argument register, as per the ABI, to be used by the callee. But in the vast majority of cases, functions take fewer arguments than the number of ABI registers. In future work, we plan to extend our analysis to examine the callee code to obtain a more accurate estimate of the actual argument register use.

About 9% of jump tables are deemed unsafe because our static analysis cannot rule out the possibility that a tainted value escapes to the caller. On further examination, we found that this mostly happens because the rdx register is considered a return value register by the ABI, but in most cases, only rax is used to communicate a return value. An analysis of callers can improve this accuracy.

---

\(^9\) For C++ exceptions to work in this configuration, the exception handling metadata will need to be rewritten to use encoded pointers. But we did not do this because this configuration is purely for measuring the performance impact of pointer encoding. Real-world applications will use the higher optimization levels discussed in the next few paragraphs. As a result of not preserving exception compatibility, omnetpp and porray fail in this configuration and were excluded from the average.

---

Changes from the camera-ready version: We have improved the safe jump table transformability analysis by adding call argument tracking. Initially, our analysis treats every argument register, as per the ABI, to be used by the callee. This approach makes CallArgs, where a tainted value (i.e., a value that is modified by the new jump table computations) is possibly passed as an argument to a called function. However, in the vast majority of cases, functions take fewer arguments than the number of ABI registers. Therefore, we classify a jump table as unsafe due to CallArgs only if the tainted value is definitely passed to a callee. To do that, we extend our analysis to compute the list of argument registers used in every path at the callee and consider only these registers as function arguments. This brings down the proportion of CallArgs cases by 10×, thereby classifying 83% of the jump tables as “safely transformable” as compared to the previous number of 55%.
We focus on the two highest optimization settings here. The results with a geometric mean of 2.9x across the subset of SPEC 2006 benchmarks they used, with a geometric mean of 29x. For larger binaries, they explain how their average converges to about 10x, which includes a 4× for their array-based implementation of translation table, and up to 5× for the new code, plus 1× for the original code. Probabilistic disassembly [33] reduces the code size increase to about 6× because it can eliminate most of the false positives of superset, thereby leading the new code to be about as large as the old code. The 4× increase for the translation table is unaffected. Note that our 3.5× size is only about half of theirs.

Although the overall size of the new binary is 3.5×, note that the old code is unlikely to be accessed at runtime. So the runtime memory footprint is about 2.5× the original.

### 6.5 Discussion

The goal of SAFER is to maximize performance while avoiding unsafe failure modes. Naturally, it would be preferable to avoid failures altogether. The benefit of our approach is that it can tolerate errors on one side — false positives in disassembly and false negatives in definite code pointer identification. This enables our analyses to be tuned to minimize the type of errors that lead to failures.

The safest approach for handling binaries that lead to failure is to apply full address translation. This increases the overhead to over 5% if applied to all binaries. A second option we have tried successfully on all of our datasets is to require the stronger function prolog checks for definite code pointers. This leads to a slightly lower overhead of 3.5%. Since only a small subset of binaries need this safer treatment, the final overhead is likely to be closer to 2% rather than 5%.

These safer options can be applied when a transformed binary fails tests. A better option is to apply them proactively by leveraging the disassembly phase to flag binaries with a non-negligible likelihood of data in code. This is a topic of our ongoing research.

### Limitations

While our prototype is able to handle many complex applications, it is still a prototype and faces scalability challenges (especially in disassembly) on very large binaries. The largest binary we have been able to handle is about 75MB (clang). Firefox’s libxul at 160MB causes failures during disassembly due to resource exhaustion.

Our encoding technique needs one unused bit in pointers.
Another benefit of patching is that jumps are far more efficient to avoid having to recognize and fix up pointers, they resort to in-place patching, which involves the insertion of jumps in apply binary instrumentation on x86 binaries of significant size. In-place instrumentation. BIRD [34] was the first work to code and (ii) it leaves libraries uninstrumented. Two main drawbacks: (i) it cannot handle position-independent instrumentation systems are lighter weight in terms of performance cost but are prone to instrumentation failures because of known challenges in disassembly and code pointer identification. Early research [30, 45] established the challenges in correct disassembly of x86 binaries. Linear disassembly tools such as objdump perform poorly on binaries containing embedded data. Recursive disassembly is not affected by embedded data, but runs into coverage problems because of the difficulty of statically predicting the targets of indirect control transfers.

Instrumentation using metadata. Many prior works [15, 19, 21, 31] rely on additional compiler-generated information that is unavailable in stripped binaries, such as the symbol table. Recent works have shown [37, 40] that exception-handling metadata can be used to accurately identify functions. Many tools such as GHIDRA [44] and JIMA [2] also rely on exception-handling metadata. The advantage of this metadata is that it is unaffected by stripping. Its downside is that it may not be available for C programs. Some C++ projects also disable the inclusion of this metadata due to its large size.

Metadata-free instrumentation. Tools such as BIRD [34], Dyninst [27], and Angr [48] rely on patterns such as function prologue signature to discover indirectly reached functions to increase disassembly coverage. Such pattern matching misses a significant amount of code (≈30% for Dyninst). Other techniques such as Ddisasm [22] and Rambler [52] rely on static analysis to identify and update pointers. However, experiments conducted by Zhang et al. [63] show that Ddisasm fails to correctly transform binaries in the presence of embedded data as it misclassifies data pointers as code pointers. Similarly, SECRET [60] relies on static analysis to identify virtual function tables and can suffer from incompleteness. To get around disassembly errors while remaining metadata agnostic, techniques such as Secondwrite [49] and PSI [61] use runtime address translation. Secondwrite leverages LLVM optimization to achieve excellent performance cost. However, it has two main drawbacks: (i) it cannot handle position-independent code and (ii) it leaves libraries uninstrumented.

In-place instrumentation. BIRD [34] was the first work to apply binary instrumentation on x86 binaries of significant size. To avoid having to recognize and fix up pointers, they resort to in-place patching, which involves the insertion of jumps in the original code to transfer control to the instrumented code. Another benefit of patching is that jumps are far more efficient than hash table lookups, or even our multiplication-based encoding. As a result, patching has been used by several researchers [31, 38]. A key challenge with patching is that jumps require several bytes of space, but there can be instances where locations very close to each other may need patching. BIRD shows how to solve this problem in the worst case, where there is just a single byte of space available, by using software interrupt (i.e., INT) instructions. But this incurs a heavy performance cost. Recently, e9Patch [18] developed a clever approach that finds a way to use jumps, as long as there are at least two bytes available. This enables their technique to work in most cases, but the corner case of just one-byte space remains. This is the reason why most binary instrumentation systems rely on rewrite-time fix-up or address translation.

Another challenge with patching based approach is that it cannot tolerate false positives in disassembly. If data is mistakenly patched, it will change the program semantics. e9Patch is focused on the techniques for patching in the presence of tight constraints on space but does not concern itself with the higher-level problems in binary instrumentation, e.g., deciding the locations that should be patched.

ARMore [16] is recent work that uses a patching-based approach but is aimed at tolerating false positives in disassembly. Their method is focused on the ARM instruction set. Since our approach is implemented only on x86, a side-by-side performance comparison is not possible. In terms of the techniques, the fixed size of ARM instructions simplifies the task of in-place patching as compared to x86. Their approach for coping with embedded data that may be mistakenly patched is to mark the code as unreadable. Attempts to read the data will hence cause a memory fault, which can be handled by a signal handler that can then obtain the correct original data value. This approach can work well if embedded data is accessed infrequently, but if there is an embedded array that is used in a loop, the performance can suffer greatly. Since the technique was not evaluated on binaries containing data, this overhead question remains unanswered.

8 Conclusions

In this paper, we presented SAFER, a static binary instrumentation system that combines good performance with applicability to a wider range of binaries. A key benefit of our approach is that it tolerates most forms of disassembly and pointer identification errors. Our evaluation shows that SAFER has a low runtime overhead of ≈2%, and can successfully instrument a wide range of binaries, including SPEC 2006 and SPEC 2017 benchmarks and many commonly used applications that together add up to over 1GB of binary code. To the best of our knowledge, ours is the first binary instrumentation work to combine the performance benefits of rewrite-time code-pointer transformation with the flexibility and compatibility benefits provided by runtime address translation. This combination is enabled by a combination of analyses for identifying definite
and possible code pointers, a new algorithm for pointer encoding, and a new technique for safe jump table instrumentation.

References

[36] Chengbin Pang, Ruotong Yu, Yaochi Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. 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 (SP)*, 2021.
Fig. 7: Instrumentation process overview

A. Implementation
SAFER is implemented in C/C++ (about 40K LoC). We outline some of the key aspects of our implementation in this section.

A.1 Disassembly
As discussed in Section 2.3, we follow a recursive disassembly. The roots for the recursive disassembly are obtained as illustrated in Figure 1. We reused the disassembly implementation from our concurrent work [42] which relies on capstone to translate a given sequence of bytes to assembly.

Static analysis for computed code pointers To obtain computed code pointers or jump table targets, we perform analysis discussed in Section 3.2. First, our disassembly output is lifted to an intermediate representation, specifically, gcc’s RTL. This is done using our architecture neutral system LISC [28]. An intra-function analysis is then performed on this IR to discover potential jump table targets. The discovered targets are once again fed into the disassembler to discover new code, which is analyzed again to find more jump tables. The process is repeated until no new jump tables are discovered.

A.2 Instrumentation
Figure 7 gives an overview of how SAFER generates a functional instrumented binary. Below are the steps involved:

Preserving direct control flow: Since instrumentation changes code location, the direct control flow transfers need to be adjusted accordingly. We do so by using a technique similar to BinCFI [62]. As shown in Figure 7, every instruction

Original code | Instrumented code |
--- | --- |
1200: mov *%rip, %eax | L1200: mov *%rip, %rax |
1209: mov *%rip, %eax | L1209: mov %rax, %fs:0x88 |
// load function pointer from location 310 | call translation routine |
120e: call %rax | L120e: call %rax, %fs:0x88 |
1211: lea -0xf18(%rip),%rdi | L1211: lea *%rip, %rdi |
1218: cmp $0x14, %rax | L1218: cmp $0x14, %rax |
121c: jge L12ff | L121c: jmp %rdi |
1226: add %rax,%rdi | L1226: add %rax, %rdi |
1229: jmp *%rdi | L1229: jmp *%rdi |
// Indirect jump using jump table |
122a: ... | L122a: ... |
1270: ... | L1270: ... |
1298: ... | L1298: ... |
L1300: ret | L1300: ret |
L1300: push %rbp | L1300: push %rbp |
1301: sub $0x20, %esp | L1301: sub $0x20, %esp |
... | ... |
Static data: | Static data: |
// Jump table... | // Rewritten jump table |
300: 0x0da | L300: long L122a-L300 |
304: 0x7f0 | L304: long L1270-L300 |
308: 0x9f8 | L308: long L1298-L300 |
// Pointer constant | |
310: 0x1300 | L310: 32byte 0x1300 |
is assigned a unique label based on its address in original binary (e.g., L1200 for instruction at 1200). The control flow transfers are then changed to use these labels instead of constant offsets. For example, the jump instruction at 121ec is modified from jge 12ff to jge L12ff. Note that this is different than Symbolization discussed in Uroboros [53]. Uroboros tries to solve the problem of identifying and fixing up code pointers, which it refers to as the Symbolization.

**Preserving indirect control flow:** SAFER either encodes possible code pointers or leaves them unchanged (Section 4). Encoding of pointers is done at load time with the help of a customized loader (Section A.3). Every indirect control flow transfer needs to be instrumented to translate the pointers. Figure 7 shows instrumentation of indirect call at 120e that saves register rax into a thread local storage (%fs:0x88), moves the indirect target into rax and jumps into a translation function. After decoding or performing address translation, this routine jumps to a trampoline that is specialized for the destination address. This trampoline restores rax register and then jumps to the ultimate target (e.g., L1300_tt for function at 1300).

**Recreating jump tables:** Our safe jump table analysis discussed in Section 5 allows us to safely recreate certain jump tables and avoid instrumentation of the corresponding indirect jumps. As shown in Figure 7, the jump table at 300 is recreated using labels of corresponding targets and the corresponding indirect jump at 1229 is left uninstrumented.

**Handling system calls.** Some system calls take function pointers as argument. If the function pointer is encoded, it must be decoded before it is passed to the kernel. Specifically, we have implemented this feature for the rt__sigaction system call. This requires checking the syscall instructions in glIBC and libpthread and instrumenting them.

**Reassembling instrumented binary:** SAFER generates an assembly file containing all the aforementioned instrumentation as well as the ELF headers (e.g., program headers, section headers, etc). This assembly code is assembled into an object file using the system assembler. The text section of this object file is the ELF image of instrumented binary which is extracted and used for program execution.

### A.3 Customized loader

To support our pointer encoding/decoding and runtime address look up, we rely on a customized loader. During instrumentation, the interpreter associated with the binary is modified to point to this customized loader. (Uninstrumented binaries continue to use the original system loader.)

We made our customizations on the default loader on Ubuntu 20.04 (GNU libc-2.31 loader ld.so). Our changes, implemented in about 1500 LoC, perform the following tasks: (1) populating the GTT, (2) handling encoded pointers, and (3) loading instrumented modules. Task (1) was already described in Sec. 4.3. For (2), the loader’s default relocation process modifies all pointers by adding a module’s base address to the pointer offset. Instead, our custom loader encodes these pointers. To initiate this, we introduced a new relocation type. When our customized loader encounters this relocation type, it performs encoding instead of the usual addition of base address.

For Task (3), we changed the loader’s default search process to always load an instrumented version of a shared library when requested by an instrumented program.

### B Supporting security-relevant instrumentation

#### B.1 Shadow stack instrumentation

Code reuse attacks such as return oriented programming (ROP) [47] rely on executing a chain of code gadgets to achieve an attacker’s goal. Shadow stack [24, 39] is a well-known defense to thwart these attacks. The key idea is to maintain a second copy of return addresses on a shadow stack, and to compare the two return addresses before executing each return instruction. Several research efforts have demonstrated that coarse-grained control-flow integrity is insufficient to defend against code-reuse attacks [13, 25, 26], and that the precision of shadow stacks is required [10].

Binary instrumentation for shadow stacks has a long history, going back to BIRD [34]. Shadow stack based protection has often been used to demonstrate binary instrumentation capabilities [4, 61]. Consequently, we chose it as an example illustration for SAFER.

Shadow stack is known to be incompatible with functionalities such as exception handling and longjmp that cause multiple frames to be popped off the stack. This will cause loss of synchronization between the shadow stack and the main stack. Several techniques have been proposed to address this, and we rely on the one proposed by Burow et al. [9] to push the current stack pointer onto the shadow stack as well.

Our instrumentation for shadow stack is shown in Fig. 8. We instrument both calls and returns. Our shadow stack pointer is stored in thread local storage (%fs:0x78). Calls are
This results in an overhead of about 20%. Our shadow stack control-flow hijack, which can be realized using an indirect call instructions, the attack can succeed. All they need is an initial instructions. Since there is no instrumentation protecting these can launch a ROP chain that uses only unintended return instructions. As attackers shadow stacks by themselves are not secure, as attackers can access a vulnerability that leaks chosen locations on the stack or heap, or large numbers of such locations, they can sufficient under some threat models. For instance, if attackers can control flow, they can create a new instruction, which can be used to control the flow of execution. This can lead to the execution of unintended code, which can be used to achieve the desired effect.

Coarse-grained CFI can also provide a limited defense against code reuse attacks on its own. But more importantly, it provides a basis to build secure instrumentation [1, 20, 61]. For instance, shadow stacks by themselves are not secure, as attackers can launch a ROP chain that uses only unintended return instructions. Since there is no instrumentation protecting these instructions, the attack can succeed. All they need is an initial control-flow hijack, which can be realized using an indirect call or a jump, which is unprotected by the shadow stack. However, by combining a coarse-grained CFI with shadow stack, we can prevent the use of any unintended instruction, thereby taking away the attacker’s ability to bypass the shadow stack.

Since we instrument every indirect jump and call, coarse-grained forward-edge CFI can be easily realized using SAFER. We consider two flavors of protection, each offering a different trade-off between security and performance. The most efficient approach is to rely on the following factors:

- Possible code pointers go through address translation. This table, in effect, enforces a coarse-grained CFI policy, as in the case of BinCFI [62].
- Definite code pointers are encoded. Because of the properties of encoding established in Sec. 4, encoding scatters the original pointer values across the entire address space. This means that the attacker has a negligible chance of guessing the encoded value of her intended target without knowing the multiplier used for encoding or decoding. By selecting the multiplier at random at load time and ensuring that it is stored in the thread-local storage, we can achieve randomized protection that is similar to how stack canaries are protected. Overall, this scheme is similar to PointGuard [11].
- Indirect jumps used in transformed jump tables are not checked, but note that by virtue of the static analysis used, we already know that the target is determined by the contents of the jump table, which resides in read-only memory. We also know that values derived from the jump table content are never stored memory (“Deref” property), so no memory corruption will allow the attacker to control the target of a transformed jump table jump.

Thus, all forward edges are protected by coarse-grained CFI even when all of the optimizations are in effect. Thus, overhead for coarse-grained forward edge CFI protection is 1.9% for SPEC 2006 compiled into PIE, and 5.2% for SPEC 2006 compiled into non-PIE.

The protection provided by encoded pointers may not be sufficient under some threat models. For instance, if attackers can access a vulnerability that leaks chosen locations on the stack or heap, or large numbers of such locations, they can

![Fig. 9: Percent runtime overhead of CFI and Shadow stack. On the left, we have binaries with runtime overhead for CFI less than 20% and on the right we have binaries with overheads higher than that.](image-url)
often find the value of an encoded pointer whose value they already know. This would allow them to extract the multiplier used for encoding. Note that this is similar to attacks that infer the XOR mask used for stack canaries or PointGuard [11]. If protection against such adversaries is desired, there are two options. One is to implement an additional checking mechanism on top of pointer encoding. The second is to simply fall back on the use of address translation for all pointers. We measured the overhead in this mode, and the results are shown in Fig. 9. The average overhead across SPEC 2006 is 5.1%.

C Artifact Appendix

C.1 Abstract

The artifact submission is for the paper titled "SAFER: Efficient and Error-Tolerant Binary Instrumentation". Our tool SAFER is an efficient and safe binary-instrumentation suite that is capable of instrumenting complex programs. The tool is compatible with both position independent (PIE) and position dependent (Non-PIE) executables and has a modest overhead of ≈2%.

The artifact consists of software and will be submitted in the form of a VM containing a pre-installed version of the software and scripts needed to run the software. It will also contain all the instrumented programs used during evaluation. SAFER was used to instrument 15 real world programs along with their shared libraries, customized coreutils binaries with data embedded in code and SPEC 2006 and 2017 binaries. The total size of all programs was about 1.1GB.

C.2 Description & Requirements

SAFER’s current prototype requires Ubuntu 20.04 operating system. It further requires additional packages (Capstone and Ocaml) for disassembly and static analysis. We are submitting our artifact as an Oracle VirtualBox VM. SAFER is already installed in the VM along with all the prerequisite packages. The VM also contains SAFER’s source code.

Requirements to run artifact: A x86-64 system with Oracle VirtualBox is required to use our artifact. Importing the virtual box image through Oracle Virtualbox running on a x86-64 system will recreate the testing environment. The virtual machine image is configured with 8GB of RAM and 100GB of secondary storage. So we recommend to run it on a system with at least 16GB of memory and 256GB of storage space.

Benchmarks like SPEC CPU 2006 and 2017 are already installed and set up with instrumented binaries. Other datasets like instrumented real-world applications and data-in-code coreutils are also present in the virtual machine.

C.2.1 Security, privacy, and ethical concerns

Since we are providing our system in a virtual machine, there is no risk for the host machine of the evaluator.

C.2.2 How to access

SAFER’s artifact http://seclab.cs.sunysb.edu/seclab/safer

C.2.3 Hardware dependencies

An x86-64 machine preferably with 16GB of RAM and 256GB of storage space.

C.2.4 Software dependencies

Oracle VirtualBox 6 is required to run the virtual machine. The system setup in the virtual machine is complete and requires no additional software installation.

C.2.5 Benchmarks

For testing the performance overhead we use SPEC CPU 2006 and SPEC CPU 2017 benchmark suites. For functionality evaluation we use 15 pre-built real-world programs and custom-compiled coreutils with data-in-code.

C.3 Set-up

C.3.1 Installation

1. Download the virtual machine image file.
2. Install and open VirtualBox.
4. Select the virtual machine image.
5. Click import.

C.3.2 Basic Test

Instrumenting a program ls with its shared libraries.

1. Start the virtual machine.
2. Login with the password ‘safer’.
3. Copy the original ls binary to the test folder.

   `> cp /usr/bin/ls ~/SBI/test`

4. Find the dependencies of ls.

   `> cd ~/SBI/testsuite`  
   `> ./nd_libs.sh /home/safer/SBI/test/ls`  
   `> truncate -s 0 randomized.dat`

5. Run instrumentation on ls.

   `> cd ${HOME}`  
   `> ./instrument_prog.sh ls`

6. Wait for the instrumentation to finish. Printed date means the process is completed.
7. Go to the output directory.

   `> cd ${HOME}/instrumented_libs/`

8. Run the instrumented ls. Contents of the directory should be printed.

   `> ./ls`
C.4 Evaluation workflow

C.4.1 Major Claims

(C1): SAFER can handle disassembly errors in the presence of complexities such as data embedded in code.

(C2): SAFER’s pointer encoding scheme can detect instrumentation errors at runtime and deterministically abort (FAIL-CRASH).

(C3): SAFER is able to instrument position dependent (Non-PIE) code.

(C4): SAFER’s pointer encoding and safe jump table transformation helps in achieving safe instrumentation with a fail-crash while having a modest overhead of $\approx 2\%$.

(C5): SAFER is able to instrument a wide variety of real world programs.

C.4.2 Experiments

(E1): [Data-in-code test]: We use coreutils and its built-in tests to ensure that SAFER is successfully able to instrument binaries where data is present in the code section.

Generating the dataset: We make use of a linker-script to compile a version of coreutils where read-only data is embedded in the code. The binaries and pre-built and available in the VM: `/home/safer/coreutils/coreutils-data/bin`.

Test preparation: Pre-instrumented binaries with SAFER’s different modes (Table 3 in the paper) are available in `/home/safer/coreutils/coreutils-data` directory as below:

- `bin_FN_PRLG`: Function prologue based pointer classification.
- `bin_FULL_AT`: Run time address translation.
- `bin_valid_ins`: Valid instruction boundary based pointer classification.
- `bin_ABI`: ABI specification based pointer classification.

To reuse the above pre-instrumented binaries, copy all the binaries from one of the above mention directories to `/home/safer/coreutils/coreutils-8.30/src/`. Alternatively, binaries can be re-instrumented as follows:

```
> cd /home/safer
> ./instrument-coreutils.sh \
  config=<FULL_AT/FN_PRLG/valid_ins/ABI>
```

Testing: Coreutils in-buit testsuite is used to test correctness of instrumented binaries:

```
> cd /home/safer/coreutils/coreutils-8.30
> make check
```

Results: FN_PRLG and FULL_AT configurations result in 100% correct instrumented binaries. There is 1 failure due to one of the test making use of LD_PRELOAD to load a dynamically built library. The library is built by the test case at the runtime and used. SAFER requires all program modules to be instrumented for correct execution. Since, this particular library is not available to us during instrumentation time, it is left unchanged and resulted in a crash.

Other approaches (valid_ins and ABI) specifications result in failures that are detected by SAFER’s fail-crash design. The test will not proceed unless the faulty binaries are replaced. The list of failed binaries is in coreutils-data/ABI_fail.sh and coreutils-data/valid_ins_fail.sh. Running these scripts will replace the faulty binaries with correctly instrumented (with FULL_AT) ones and the test can progress.

(E2): [Non-PIE binaries]: SPEC 2006 binaries are compiled as non-PIE and then instrumented. Successful completion of SPEC tests ensure correct transformation. Other non-PIE programs such as gcc and Python have also been tested.

How to: We are providing the instrumented non-PIE SPEC binaries to reduce the testing time. However, if you want to instrument then again follow the steps in the preparation section otherwise skip to the execution section.

Preparation: Clean up and start the instrumentation.

```
> ./instrument-suite.sh \
/home/safer/spec-06/nopie
```

Execution: Change the directory to the spec-06 and run the command.

```
> cd /home/safer/spec-06/
> source shrc
> runspec --config=nopie.inst.cfg \ 
  --noreportable \ 
  --iterations=1 all
```

Wait for the SPEC run to complete. This may take several hours to complete ($\approx 4$ hours).

- For real world programs (gcc and python), the testing process is described in the subsequen section.

Results: After the SPEC run is completed check the log file mentioned. There should be results for all the binaries except wrf and gamess whose uninstrumented versions fail on our setup.

(E3): [Runtime overhead with SPEC 2006]

How to: SPEC 2006 CPU benchmark suite is used to test the runtime overhead caused by SAFER’s instrumentation. Note that the below steps are time consuming. Hence, they should be run in background (e.g., using nohup). Furthermore, we are providing pre-instrumented binaries. Hence, the preparation step can be skipped.

Preparation: Instrument the SPEC binaries.

```
> truncate -s 0 
/home/safer/SBI/testsuite/randomized.dat
> cd /home/safer/
> ./instrument-suite.sh \
  /home/safer/spec-06/pie/
```

Wait for the instrumentation to complete.
**Execution:** Run the testsuite with uninstrumented binaries (base result).

```bash
> cd /home/safer/spec-06/
> source .shrc
> runspec --config=default.cfg \
    --noreportable all
```

Wait till completion and save the result directory as base. Then, run the instrumented binaries:

```bash
> cd /home/safer/spec-06
> runspec --config=inst.cfg \
    --noreportable all
```

Save the instrumented version results.

**Results:** We computed the overhead by importing the corresponding csv files onto a spreadsheet and comparing the timings of instrumented run over base run. We are also providing the results of our experiments that have been published in the paper ([/spec-06/our_results](/spec-06/our_results)).

(E4): [Performance vs optimization levels] We compiled SPEC CPU 2017 benchmarks at 6 optimization levels (O0, O1, O2, O3, Ofast, and Os) and measured the runtime overhead for each of them.

**Preparation:** Running the uninstrumented binaries:

```bash
> cd /home/safer/spec-17/
> source shrc
> runcpu --config=defaultO0.cfg \
    --noreportable in tspeed
> runcpu --config=defaultO0.cfg \
    --noreportable fpspeed
```

**Execution:** Running the instrumented binaries:

```bash
> cd /home/safer/spec-17
> runcpu --config=default.inst.O0.cfg \
    --noreportable intspeed
> runcpu --config=default.inst.O0.cfg \
    --noreportable fpspeed
```

**Results:** Other optimizations could be tested by repeating the steps above with O0 replace by either O1, O2, O3, Ofast, or Os. The results can be obtained in the same manner as SPEC 2006.

(E5): [Safe jump table transformation]: SAFER’s safe jump table analysis (Section 5 in paper), helps in improving performance while maintaining correctness of instrumentation by avoiding instrumentation of indirect jumps related to jump tables.

**Preparation:** Safe jump table analysis code is present in [/home/safer/SEF/safe_jtable]. Steps to build this code are present in a README file in the same directory.

**Execution:** Safe jump table results (Figure 5 in paper) were produced on SPEC 2006 binaries. Please follow the steps in README to reproduce the results.

(C.5) **Version**

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at [https://secartifacts.github.io/usenixsec2023/](https://secartifacts.github.io/usenixsec2023/).