SAFER: Efficient and Error-Tolerant Binary Instrumentation

Soumyakant Priyadarshan, Huan Nguyen, Rohit Chouhan, and R. Sekar, *Stony Brook University*

https://www.usenix.org/conference/usenixsecurity23/presentation/priyadarshan

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

SAFER: Efficient and Error-Tolerant Binary Instrumentation[*]

Soumyakant Priyadarshan, Huan Nguyen, Rohit Chouhan, and R. Sekar
Stony Brook University, NY, USA.
{spriyadarsha, hnnguyen, rchouhan, sekar}@cs.stonybrook.edu

Abstract

Recent advances in binary instrumentation have been focused on performance. By statically transforming the code to avoid additional runtime operations, systems such as Egalito and RetroWrite achieve near zero overheads. The safety of these static transformations relies on several assumptions: (a) *error-free* and *complete* disassembly, (b) exclusive use of position-independent code, and (c) code pointer identification that is free of both *false positives* and *false negatives*. Violations of these assumptions can cause an instrumented program to crash, or worse, experience delayed failures that corrupt data or compromise security. Many earlier binary instrumentation techniques (e.g., DynamoRio, Pin, and BinCFI) minimized such assumptions, but the price to be paid is a much higher overhead, especially for indirect-call-intensive (e.g., C++) applications. Thus, an open research question is whether the safety benefits of the earlier works can be combined with the performance benefits of recent works. We answer this question in the affirmative by presenting a new instrumentation technique that (a) *tolerates* the use of *position-dependent code* and common disassembly and static analysis errors, and (b) *detects* assumption violations at runtime *before* they can lead to undefined behavior. Our approach provides a *fail-crash* primitive for graceful shutdown or recovery. We achieve safe instrumentation without sacrificing performance, introducing a low overhead of about $\sim 2\%$.

1 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], RETROWRITE [17] and SBR [41] that achieve very good performance but make optimistic assumptions

[*]This work was supported by an ONR grant N00014-17-1-2891 and in part by NSF grants 1918667 and 2153056.

about the binary. Specifically, the last three works assume that:

- there is no data interspersed with code,
- the binary is a 64-bit position-independent binary, and
- all jump tables[1] in the binary follow one of the expected patterns that have been programmed into these systems.[2]

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 *SAFe 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.* 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[3] and has the following goals:

- *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.
- *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.
- *Incur low overhead.* We aim to achieve performance overheads close to 2% on average.
- *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 ($\approx 5\%$) while incurring low overheads ($\approx 2\%$) 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

[1] Jump 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.

[2] CCFIR [58] makes similar asmumptions for x86-32 binaries on Windows.

[3] Although C++ exception unwinding metadata is not stripped from deployed binaries, we don't rely on this metadata for disassembly or instrumentation. This is because many C++ projects don't use exceptions and disable the inclusion of this metadata due to its bulkiness (e.g., Chrome). Moreover, on platforms other than Linux/x86, this metadata is limited to C++ programs and hence can't be used for instrumenting C and assembly code.

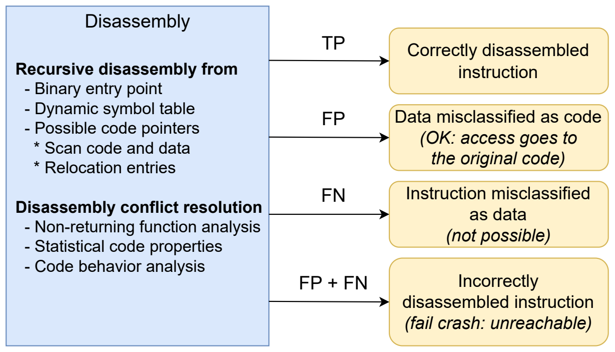
Disassembly
Recursive disassembly from
- Binary entry point
- Dynamic symbol table
- Possible code pointers
* Scan code and data
* Relocation entries
Disassembly conflict resolution
- Non-returning function analysis
- Statistical code properties
- Code behavior analysis
TP
FP
FN
FP + FN
Correctly disassembled instruction
Data misclassified as code (OK: access goes to the original code)
Instruction misclassified as data (not possible)
Incorrectly disassembled instruction (fail crash: unreachable)

Fig. 1: Our disassembly involves recursive disassembly followed by a conflict resolution. The impact of false positives (FP) and false negatives (FN) is displayed on the right.

in Sec. 3. We describe our pointer encoding and safe jump table transformation techniques in Sec. 4 and 5 respectively. Experimental evaluation is presented in Sec. 6, followed by related work discussion in Sec. 7 and concluding remarks in Sec. 8.

2 Tolerating Disassembly Errors

Accurate disassembly of x86 binaries has long remained a challenge. A recent survey by Pang et al. [36] shows that every state-of-the-art disassembly tool experiences non-zero false negatives and false positives across all compilers and optimization levels. This means that even with the best disassembly tool, a straightforward application of instrumentation will break some binaries. Our strategy for tolerating these errors follows that of previous works in Group I. It is summarized in Fig. 1 and described further below.

2.1 Tolerating false positives in disassembly

Disassembly false positives can affect instrumentation correctness in two ways. First, data that is identified as code may be instrumented, which will cause it to change. When the instrumented program uses this modified data, its behavior will be changed. Second, if instrumented execution ever reaches this data, the result would be unpredictable as well.

Our approach for tolerating both these errors is similar to that of previous Group I techniques that instrument a second copy of code, while preserving the original contents in a read-only section [7, 32, 62]. Thus, any access to this data will return the exact same values as in the uninstrumented program. It can be easily seen that the second type of error is not possible, as a legitimate program will never jump into its data.

Binary stirring [54] and Reins [55] use the same basic strategy of leaving the original code in place, but then modify some of its contents in order to significantly speed up address translation over Group I techniques. Specifically, any location in the original code that is determined to be a possible indirect branch target by their static analysis is modified this way. As a result, false positives have the potential to break their approach. For this reason, we have not included these two works in Group I.

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

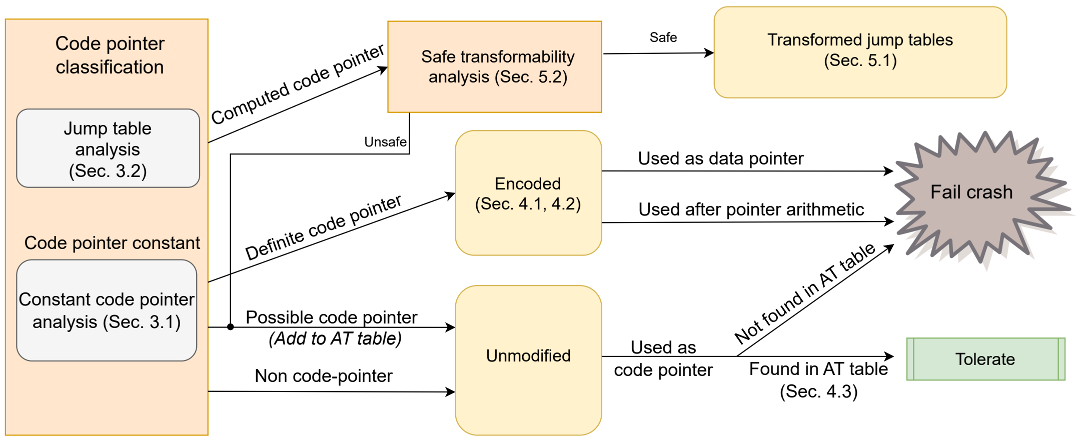
Code pointer classification
Jump table analysis (Sec. 3.2)
Code pointer constant
Constant code pointer analysis (Sec. 3.1)
Computed code pointer
Definite code pointer
Possible code pointer (Add to AT table)
Non code-pointer
Safe transformability analysis (Sec. 5.2)
Unsafe
Safe
Transformed jump tables (Sec. 5.1)
Encoded (Sec. 4.1, 4.2)
Used as data pointer
Used after pointer arithmetic
Fail crash
Unmodified
Used as code pointer
Not found in AT table
Found in AT table (Sec. 4.3)
Tolerate

Fig. 2: High-level overview of our approach.

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]
- return addresses pushed on the stack.[5]

(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

[4]Compilers generate this instruction when high-level code takes the address of a function.

[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:* $Target = Base + Table[index * stride]$
- *Pattern 2:* $Target = Table[index * stride]$
- *Pattern 3:* $Target = Base + index * stride$

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 tables. 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,

		Number of indirect jumps using a jump table	Percentage
Groundtruth		60165	100%
Detected	Pattern 1	16102	26.76%
	Pattern 2	43957	73.06%
	Pattern 3	4	0.01%
Missed by the 3 patterns		102	0.17%

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

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*[6] — 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.

[6]This 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:

```
if $encoded(target)$ then
    $target \leftarrow decode(target)$
else
    $target \leftarrow address\_translate(target)$
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 $AB \equiv 1 \ \mathbf{mod}\ 2^n$.*

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:

$$A, 3A, 5A, \ldots, (2^n - 1)A$$

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

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

Since A, X and Y are all odd, they have no common factors with 2^n. So the only way for $A(X-Y) \equiv 0 \ \mathbf{mod}\ 2^n$ to hold is if $X \equiv Y \ \mathbf{mod}\ 2^n$. 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 $1, 3, 5, \ldots 2^n - 1$. Thus, there must be some odd number B such that $AB \equiv 1 \ \mathbf{mod}\ 2^n$. ■

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 $1 \ldots 2^{n-1}$.
    2. Compute $B$ such that $AB \equiv 1 \ \mathbf{mod}\ 2^{n-1}$.
function $enc(x) = (Ax \ \mathbf{mod}\ 2^{n-1}) \mid 2^{n-1}$
function $dec(x) = B \cdot (x \,\&\, (2^{n-1} - 1)) \ \mathbf{mod}\ 2^{n-1}$
```

Note that $x \ \mathbf{mod}\ 2^k$ 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 $y = O \cdot 2^r$ where O is odd, then $dec(y)$ is uniformly random over* $\{2^r, 3 \cdot 2^r, 5 \cdot 2^r, \ldots, (2^{n-1-r} - 1) \cdot 2^r\}$.

The first point follows from the definition above:

$$dec(enc(x)) \equiv B(Ax) \equiv (BA)x \equiv x \ \mathbf{mod}\ 2^{n-1}$$

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 $\{2^r, 3 \cdot 2^r, 5 \cdot 2^r, \ldots, (2^{n-1-r} - 1) \cdot 2^r\}$.[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,* $Pr[valid(dec(y))] = S/2^{n-1}$.

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 $S/2^r$ that will have r bits of alignment, the likelihood that $B \cdot y$ is one of these valid addresses is:

$$\frac{S/2^r}{2^{n-r-1}} = \frac{S}{2^{n-1}} \qquad \blacksquare$$

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 $1 - S/2^{n-1}$. 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_o, O_n) to the LTT, where O_o is the offset of P in the original code, and O_n 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

[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.

to fall back on. This requires updating the jump table contents so that it now stores offsets in the *instrumented* code rather than the *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 [36], Angr [48], Dyninst [5], and Ghidra [44] respectively miss about 24%, 1%, and 8% of the jump table targets. Egalito [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 [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 $Base + Table[index]$ will be transformed into one that accesses $NewBase + NewTable[index]$, where $NewTable$ is the new location of the jump table. Contents of this relocated jump table will be changed so that the $NewBase + NewTable[index]$ will point to the new location of the code accessed by $Base + Table[index]$ for each value of *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, *Base* and *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:

- *Deref:* Tainted value should not be used to dereference memory except (a) the intended memory read, namely, the instruction that originally read $Table[index]$, and (b) the indirect jump used by this jump table.
- *StoreMem:* Tainted value should never be stored in memory.
- *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 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 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 `gamess` 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.

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

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

coreutils	Rewrite success	Execution success	Safe Failure
runtime AT	105/105	105/105	NA
Instruction boundary	105/105	43/105	62
ABI validation	105/105	74/105	31
Function prologue matching	105/105	105/105	NA

Table 3: Coreutils with data embedded in code.

runtime address translation. The first row (runtime AT) in Table 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 of 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 applications 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 encounter crashes and need to initiate reinstrumentation.

Position-dependent (non-PIE) binaries. We first recompiled SPEC 2006 into position-dependent code (non-PIE). Note that Group II approaches such as Egalito [57], RETROWRITE [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 techniques 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

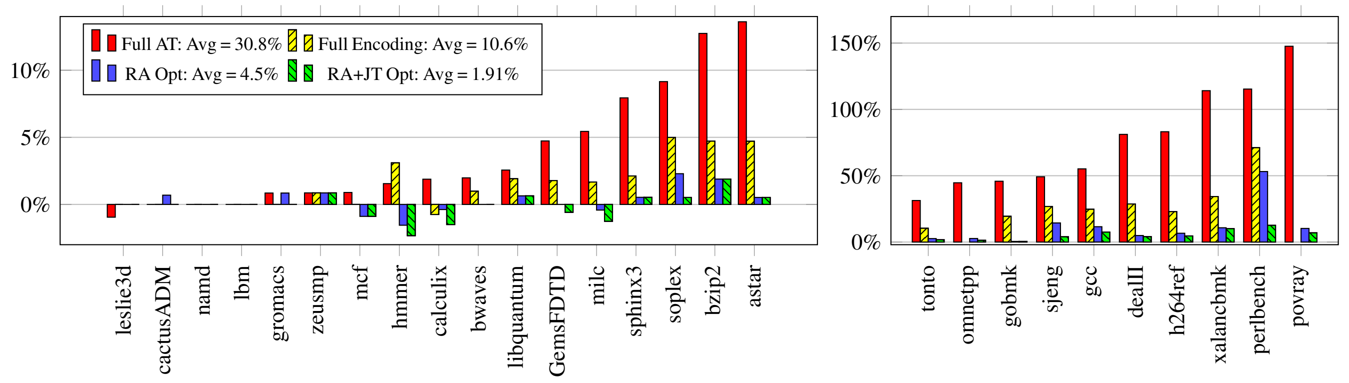

Fig. 3: Runtime overhead for SPEC 2006 compiled into position-independent executable (PIE) with different SAFER optimizations enabled. To improve readability, these results have been grouped into two charts, the left one with lower overhead benchmarks and the right including the rest.

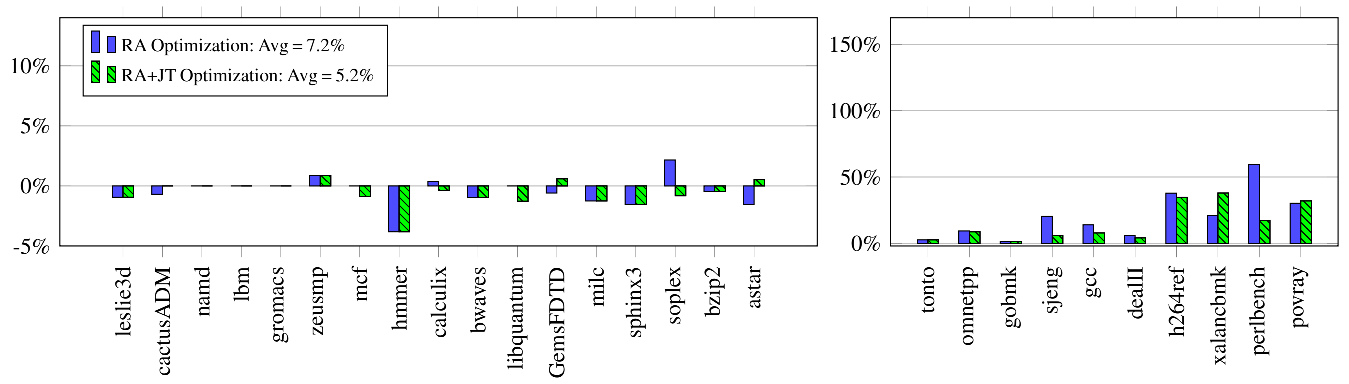

Fig. 4: Runtime overhead for non-PIE binaries. Only the two highest levels of SAFER optimization are shown.

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.

Benchmark	**O0**	**O1**	**O2**	**O3**	**Ofast**	**Os**
perlbench	3.61	25.31	7.05	8	5.92	17.93
gcc	10.97	2.54	3.4	2.27	2.64	1.49
mcf	5.31	6.21	6.25	6.46	7.95	6.34
omnetpp	4.58	11.54	12.05	10.89	11.6	12.23
xalancbmk	1.47	3.64	5.59	5	5.07	5.23
x264	-2.05	-0.42	-1.65	-1.83	-2.69	-0.41
deepsjeng	0.48	-0.4	0.84	0.43	1.3	-0.77
leela	-0.1	-0.61	0.63	0.68	0.68	-0.59
exchange2	0.82	3.13	2.7	0	2.8	1.37
xz	0.19	0.14	0.44	0.44	0.44	0.28
bwaves	0	0.13	0	0.13	NA	0
cactuBSSN	0.71	0.64	-1.27	0.63	0.63	0.6
lbm	-0.38	0.13	-0.13	-0.38	0	0.13
wrf	0.12	0.88	0.84	0.34	NA	0.46
pop2	-0.41	-1.28	1.09	-0.22	0	0
imagick	0.09	0.21	0	-0.54	-0.34	0
nab	1.93	7.19	2.44	1.22	2.07	4.44
fotonik3d	0.59	0.96	0.66	0.66	0.67	0.99
roms	0.53	1.18	0.95	0.96	NA	0.46
Average	**1.5**	**3.22**	**2.2**	**1.85**	**2.42**	**2.64**

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 eases 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%.

[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 `povray` fail in this configuration and were excluded from the average.

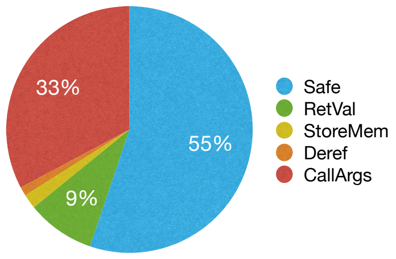

Fig. 5: The "Safe" area shows the percentage of jump tables that are transformed safely by SAFER in SPEC 2006 binaries (optimization O2). The rest are classified as unsafe, with the labels corresponding to the reasons listed in Sec. 5.2.

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.

6.3.5 *Performance on position-dependent code (non-PIE)*

Finally, Fig. 4 shows the overhead of our approach for non-PIEs. We focus on the two highest optimization settings here. The overheads are generally higher for non-PIEs: in the absence of relocation information, our analysis classifies most pointers as *possible* rather than *definite*, thus subjecting them to address translation. As a result, the overhead for the RA optimization setting increases to 7.2% as compared to 4.5% for PIEs.

The addition of safe jump table optimization brings the overhead to 5.2%. Note that this is about half the overhead reported by previous methods that can support non-PIEs such as BinCFI [62] and Multiverse [4].

6.4 Memory Overhead

Fig. 6 shows the size increase of SPEC 2006 binaries after they are rewritten by SAFER. Recall that we leave the original code in place, so that contributes an additional 1× to the size of the

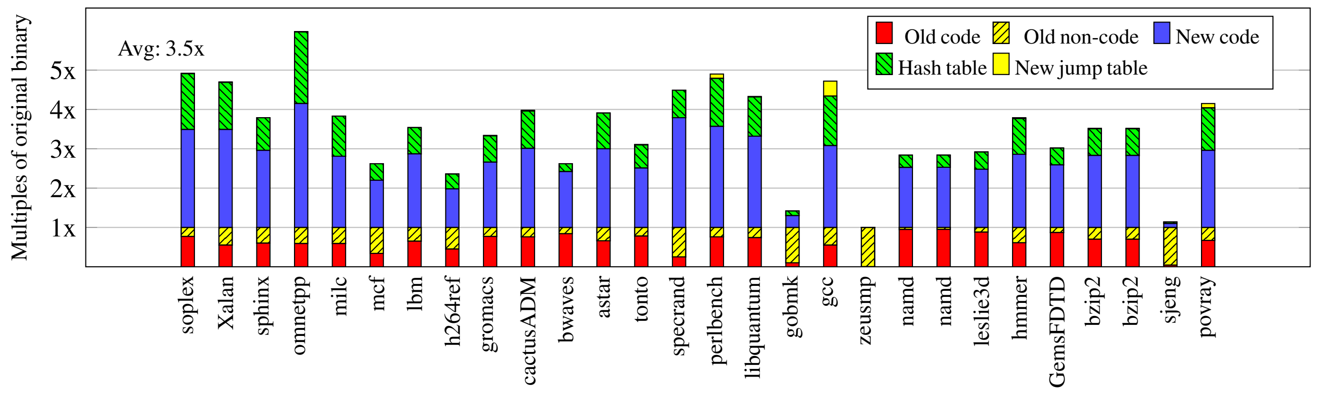

Fig. 6: New code and hash table size overhead of SPEC binaries as a multiple of original/old code.

instrumented binary. Then the instrumented code is added. Because we instrument every indirect control transfer and return, among other things, this increase is noticeable: on average, the new code is about 1.7× the size of the original binary.

The second major contribution is from the hash tables. Our possible code pointer analysis is conservative and significantly overestimates possible code pointers. Moreover, for compatibility reasons, we add return addresses to the translation table. And finally, for efficiency, we use a load factor of 25% for the hash tables. As a result of these measures, the average size of hash tables in binaries is about 0.7× the original binary size. Additionally, we also recreate jump tables as part of safe jump table instrumentation. These new jump tables are added as additional read-only data. However, they do not contribute much to the overall memory overhead. The average size of the new jump tables is 0.02× the original binary size.

Altogether, instrumented binaries are about 3.5× the size of the original binaries. While this is significant, it is far less than that reported by other Group I techniques. Note that BinCFI does not report memory overhead, so we cannot compare our results with it. Multiverse reports an increase between 5.4× and 203× across the subset of SPEC 2006 benchmarks they used, with a geometric mean of 29×. For larger binaries, they explain how their average converges to about 10×, 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. Programs that repurpose all unused bits of code pointers pose a challenge. We plan to develop a workaround based on setting aside a fraction of the address space in the loader.

7 Related Work

Dynamic binary instrumentation systems such as DynamoRIO [7], Pin [32], Valgrind [35] and Strata [46] are not prone to instrumentation errors but incur heavy performance penalty on real-world applications. Static binary 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 Ramblr [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 $\approx 2\%$, 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

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM TISSEC*, 2009.

[2] Jim Alves-Foss and Jia Song. Function boundary detection in stripped binaries. In *ACSAC*, 2019.

[3] Michael Backes and Stefan Nürnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *USENIX Security Symposium*, 2014.

[4] Erick Bauman, Zhiqiang Lin, and Kevin W Hamlen. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *NDSS*, 2018.

[5] Andrew R. Bernat, Kevin Roundy, and Barton P. Miller. Efficient, sensitivity resistant binary instrumentation. In *ISSTA*, 2011.

[6] Edson Borin, Cheng Wang, Youfeng Wu, and Guido Araujo. Software-based transparent and comprehensive control-flow error detection. In *Code Generation and Optimization*, 2006.

[7] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Code Generation and Optimization*, 2003.

[8] D. Brumley, I. Jager, T. Avgerinos, and E. Schwartz. Bap: a binary analysis platform. In *Computer Aided Verification*, 2011.

[9] Nathan Burow, Xinping Zhang, and Mathias Payer. Sok: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 985–999. IEEE, 2019.

[10] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security Symposium*, 2015.

[11] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointguardTM: protecting pointers from buffer overflow vulnerabilities. In *USENIX Security Symposium*, 2003.

[12] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *NDSS*, 2015.

[13] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security*, 2014.

[14] Lucas Vincenzo Davi, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and arm. In *ACM CCS*, 2013.

[15] Bjorn De Sutter, Bruno De Bus, and Koen De Bosschere. Link-time binary rewriting techniques for program compaction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2005.

[16] Luca Di Bartolomeo, Hossein Moghaddas, and Mathias Payer. Armore: Pushing love back into binaries.

[17] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting COTS binaries for fuzzing and sanitization. In *IEEE Symposium on Security and Privacy*, 2020.

[18] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. Binary rewriting without control flow recovery. In *PLDI*, 2020.

[19] Andrew Edwards, Amitabh Srivastava, and Hoi Vo. Vulcan: Binary transformation in a distributed environment. Technical report, Technical Report MSR-TR-2001-50, Microsoft Research, 2001.

[20] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. Xfi: Software guards for system address spaces. In *Operating systems design and implementation*, 2006.

[21] Alan Eustace and Amitabh Srivastava. Atom: A flexible interface for building high performance program analysis tools. In *USENIX*, 1995.

[22] Antonio Flores-Montoya and Eric Schulte. Datalog disassembly. In *USENIX Security*, 2020.

[23] Agner Fog. The microarchitecture of intel, amd and via cpus: An optimization guide for assembly programmers and compiler makers. *Copenhagen University College of Engineering*, 2, 2012.

[24] Michael Frantzen and Michael Shuey. Stackghost: Hardware facilitated stack protection. In *USENIX Security Symposium*, 2001.

[25] Enes Göktas, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *2014 IEEE Symposium on Security and Privacy*, 2014.

[26] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *USENIX Security*, 2014.

[27] Laune C Harris and Barton P Miller. Practical analysis of stripped binary code. *ACM SIGARCH*, 2005.

[28] Niranjan Hasabnis and R Sekar. Lifting assembly to intermediate representation: A novel approach leveraging compilers. In *ASPLOS*, 2016.

[29] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W Davidson. ILR: Where'd my gadgets go? In *IEEE Security and Privacy*, 2012.

[30] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *USENIX security Symposium*, 2004.

[31] Michael A Laurenzano, Mustafa M Tikir, Laura Carrington, and Allan Snavely. Pebil: Efficient static binary instrumentation for linux. In *Performance Analysis of Systems & Software (ISPASS)*, 2010.

[32] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Programming language design and implementation*, 2005.

[33] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. Probabilistic disassembly. In *IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019.

[34] Susanta Nanda, Wei Li, Lap-Chung Lam, and Tzi-cker Chiueh. Bird: Binary interpretation using runtime disassembly. In *Code Generation and Optimization*, 2006.

[35] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[36] Chengbin Pang, Ruotong Yu, Yaohui 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.

[37] Chengbin Pang, Ruotong Yu, Dongpeng Xu, Eric Koskinen, Georgios Portokalidis, and Jun Xu. Towards optimal use of exception handling information for function detection. In *DSN*, 2021.

[38] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Security and Privacy*, 2012.

[39] Manish Prasad and Tzi-cker Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *USENIX Annual Technical Conference, General Track*, 2003.

[40] Soumyakant Priyadarshan, Huan Nguyen, and R. Sekar. On the impact of exception handling compatibility on binary instrumentation. In *ACM FEAST*, 2020.

[41] Soumyakant Priyadarshan, Huan Nguyen, and R. Sekar. Practical fine-grained binary code randomization. In *ACSAC*, 2020.

[42] Soumyakant Priyadarshan, Huan Nguyen, and R. Sekar. Accurate disassembly of complex binarieswithout use of compiler metadata. In *ASPLOS (To appear)*, 2024.

[43] Rui Qiao and R Sekar. A principled approach for function recognition in COTS binaries. In *Dependable Systems and Networks (DSN)*, 2017.

[44] Roman Rohleder. Hands-on ghidra-a tutorial about the software reverse engineering framework. In *Proceedings of the 3rd ACM Workshop on Software Protection*, 2019.

[45] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. Disassembly of executable code revisited. In *Working Conference on Reverse Engineering*, 2002.

[46] Kevin Scott and Jack Davidson. Strata: A software dynamic translation infrastructure. In *IEEE Workshop on Binary Translation*, 2001.

[47] Hovav Shacham et al. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM CCS*, 2007.

[48] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *Security and Privacy (SP)*, 2016.

[49] Matthew Smithson, Khaled ElWazeer, Kapil Anand, Aparna Kotha, and Rajeev Barua. Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness. In *Working Conference on Reverse Engineering (WCRE)*, 2013.

[50] Victor Van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive cfi. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

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

[52] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making reassembly great again. In *NDSS*, 2017.

[53] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable disassembling. In *USENIX Security*, 2015.

[54] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *ACM CCS*, 2012.

[55] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Securing untrusted code via compiler-agnostic binary rewriting. In *ACSAC*, 2012.

[56] Wikipedia. Extended euclidean algorithm. `https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm`, last accessed: 2023-01-25.

[57] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P Kemerlis. Egalito: Layout-agnostic binary recompilation. In *ASPLOS*, 2020.

[58] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *IEEE Security and Privacy*, 2013.

[59] Mingwei Zhang. *Static Binary Instrumentation with Applications to COTS Software Security*. PhD thesis, The Graduate School, Stony Brook University: Stony Brook, NY., 2015.

[60] Mingwei Zhang, Michalis Polychronakis, and R Sekar. Protecting COTS binaries from disclosure-guided code reuse attacks. In *ACSAC*, 2017.

[61] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R Sekar. A platform for secure static binary instrumentation. *ACM VEE*, 2014.

[62] Mingwei Zhang and R Sekar. Control flow integrity for COTS binaries. In *USENIX Security*, 2013.

[63] Zhuo Zhang, Wei You, Guanhong Tao, Yousra Aafer, Xuwei Liu, and Xiangyu Zhang. Stochfuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021.

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.

Original code		Instrumented code	
1200: lea	0xf7(%rip), %rdi	L1200: lea	L1300(%rip), %rdi
1209: mov	-0xefe(%rip), %rbx	L1209: mov	L310(%rip), %rbx
// load function pointer from location 310			
120e: call	*%rax	L120e: mov	%rax, %fs:0x88
		mov	%rbx, %rax
		call	translation_routine
1211: lea	-0xf18(%rip),%rdi	L1211: lea	L300(%rip), %rdi
1218: cmp	$0x14, %rax	L1218: cmp	$0x14, %rax
121c: jge	12ff	L121c: jge	L12ff
121e: add	%rdi,%rax	L121e: add	%rdi,%rax
1221: mov	(%rax), %rax	L1221: mov	(%rax), %rax
1226: add	%rax,%rdi	L1226: add	%rax,%rdi
1229: jmp	*%rdi	L1229: jmp	*%rdi
// Indirect jump using jump table			
122a: ···	// code for jmp table entry 1	L122a: ...	
1270: ···	// code for jump table entry 2	L1270: ...	
1298: ···	// code for jump table entry 3	L1298: ...	
12ff: ret		L12ff: ret	
		L1300_tt: mov	%fs:0x88,%rax
		jmp	.L1300
1300: push	%rbp	L1300: push	%rbp
1301: sub	$0x20, %rsp	L1301: sub	$0x20, %rsp
...		...	
Static data:		***Static data:***	
// Jump table...		// Rewritten jump table	
300: 0xf2a		L300: .long	L122a-L300
304: 0xf70		L304: .long	L1270-L300
308: 0xf98		L308: .long	L1298-L300
// Pointer constant			
310: 0x1300		L310: .8byte	0x1300

Fig. 7: Instrumentation process overview

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 intermidiate 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 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 `121c` 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.

```
cmp $0,%fs:0x78
jne .push_ra
call .init_shstk
.push_ra:
sub $16,%fs:0x78
push %rax
mov %fs:0x78,%rax
push %rbx
mov 16(%rsp),%rbx
mov %rbx,-8(%rax)
lea 16(%rsp),%rbx
mov %rbx,-16(%rax)
pop %rbx
pop %rax
```

call instrumentation

```
push %rdx
mov %rax,%fs:0x88
mov %fs:0x78,%rax
mov %rsp,%rdx
add $8,%rdx
.loop:
add $16,%rax
mov %rax,%fs:0x78
cmp 0(%rax),%rdx
jne .loop
mov 8(%rsp),%rdx
cmp 8(%rax),%rdx
jne abort
pop %rdx
mov %fs:0x88,%rax
ret
```

return instrumentation

Fig. 8: Shadow stack instrumentation.

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 instrumented to push both the current stack pointer and return address onto the shadow stack. Returns are instrumented to keep incrementing the shadow stack until a match for the current stack pointer is found. By continuing to pop the shadow stack until the stack pointer value matches, we can resynchronize after such non-standard returns. Once a matching stack pointer is found, the return addresses are compared. In case of a mismatch, the program is aborted.

The performance of shadow stack is shown in Fig. 9. Our primary goal was to demonstrate the functionality of

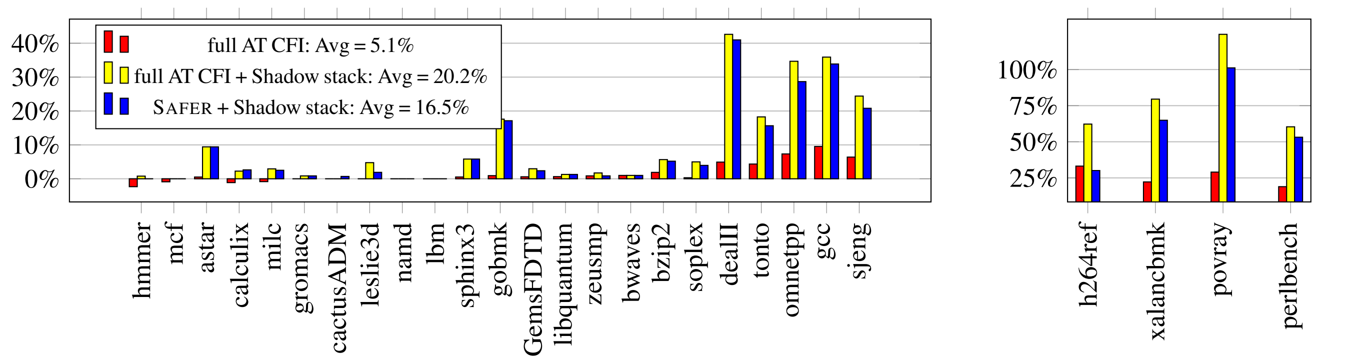

Fig. 9: Percent runtime overhead of CFI and Shadow stack. On the left, we have binaries with runtime overhead for CFI less than 20% and on the right we have binaries with overheads higher than that.

SAFER, so we did not focus on optimizing the instrumentation. Issues such as minimizing the number of register saves and simplifying the frequently executed path have not been addressed. Our design choice to store both the stack pointer and the return address on the shadow stack also extracts some performance. Consequently, the overhead of shadow stack is nontrivial at 16.5%. Shadow stack combined with CFI is more secure and can prevent from exploitation of unintended returns. This results in an overhead of about 20%. Our shadow stack overhead is comparable to that implemented by Multiverse [4] and PSI [61] as part their evaluation. Multiverse's shadow stack is incompatible with stack unwinding and works only on a subset of SPEC binaries (12 out of 27). It has an overhead of approximately 72%. On the same subset of binaries, SAFER's shadow stack has 27% overhead. PSI's implementation does not push the current shadow stack pointer on to the shadow like we do. However, it keeps popping the shadow stack untill a match is found. PSI's implementation also works on a subset of SPEC binaries (15 out of 27) and has an overhead of 18%. For the same set of binaries, our overhead is 16.2%.

B.2 Coarse-grained forward-edge CFI

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 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%.