Scalable, Sound, and Accurate Jump Table Analysis*

Huan Nguyen
Stony Brook University
Stony Brook, NY, USA
hnnguyen@cs.stonybrook.edu

Soumyakant Priyadarshan
Stony Brook University
Stony Brook, NY, USA
spriyadarsha@cs.stonybrook.edu

R. Sekar
Stony Brook University
Stony Brook, NY, USA
sekar@cs.stonybrook.edu

Abstract

Jump tables are a common source of indirect jumps in binary code. Resolving these indirect jumps is critical for constructing a complete control-flow graph, which is an essential first step for most applications involving binaries, including binary hardening and instrumentation, binary analysis and fuzzing for vulnerability discovery, malware analysis and reverse engineering. Existing techniques for jump table analysis generally prioritize performance over soundness. While lack of soundness may be acceptable for applications such as decompilation, it can cause unpredictable runtime failures in binary instrumentation applications. We therefore present SJA, a new jump table analysis technique in this paper that is sound *and* scalable. Our analysis uses a novel abstract domain to systematically track the "structure" of computed code pointers without relying on syntactic pattern-matching that is common in previous works. In addition, we present a bounds analysis that efficiently and losslessly reasons about equality and inequality relations that arise in the context of jump tables. As a result, our system reduces miss rate by 35× over the next best technique. When evaluated on error rate based on F1-score, our technique outperforms the best previous techniques by 3×.

CCS Concepts

• **Theory of computation → Program analysis**; • **Software and its engineering → Software reverse engineering**.

Keywords

static analysis, reverse engineering

1 Introduction

Static analysis of binary code plays a central role in software security [14, 24, 33, 34, 38, 41], performance optimization [21, 28], software

---

---

reliability [17], binary instrumentation [5, 11, 25, 30, 35, 37, 40], as well as reverse engineering [1, 2, 7, 8, 10, 27].

Binary analysis tools face a daunting set of challenges, including the large size and complexity of instruction sets such as the x86, and the low-level nature of instruction semantics, with numerous side effects. Compounding this further, many analyses need to be performed even before disassembly is finalized and/or the control-flow graph is constructed. In particular, many key analyses such as jump table analysis and function entry identification result in the discovery of new code or data [18, 23]. Such analyses need to be repeated on the updated control-flow graph many times until the graph converges. To cope with these challenges, binary analysis tools (e.g., Dyninst [11, 18], angr [32], Ghidra [30] and Datalog Disassembly [10], as well as many other systems based on them [5, 14, 16, 28, 35]), have been driven by pragmatic considerations such as precision and performance on real-world binaries. Soundness and applicability to all programs emphasized in static analysis research and compilers, have not been as much of a priority in these tools. Indeed, for applications such as decompilation or bug-finding, practical effectiveness is more important than the loss of soundness in some cases. However, the calculus is very different in binary instrumentation applications such as binary hardening and software debloating, where functionality preservation and applicability to all programs are paramount.

Jump table analysis, a crucial component in many existing tools [1, 2, 4, 5, 8, 10, 11, 14, 18, 21, 23, 24, 26–28, 30, 32–35, 37, 39–41, 41], is a prime example of where these differences manifest. Note that jump tables result typically from the translation of high-level language constructs such as the switch statements in C/C++. The result is an indirect jump whose target is obtained by indexing into a table stored within the read-only data of a binary. The index value itself is an expression.

A sound jump table analysis ensures that all possible targets of an indirect jump are identified. This will ensure that the control-flow graph (CFG) is complete, without any missing edges. Complete CFGs are desirable in most binary analysis applications. Without a complete CFG, some of the code in a binary won't be recognized, leading to incomplete code discovery, incomplete instrumentation, and unsound inferences about code behavior. Note that for indirect *calls*, accurately reasoning about their possible targets is very challenging. However, they can be handled conservatively because all valid function pointer values appear within the code or in the read-only data section. By scanning for these constants, we can obtain a superset of the targets [41]. In contrast, jump table pointers are the result of computation, so their values cannot be determined unless the analysis is able to identify the exact logic involved in the computation. This make jump table analysis challenging.

Most jump table analyses first compute a backward slice [36] of the code with respect to the variable (register) used in the indirect jump. Then, this slice is analyzed to uncover the base of the jump

table, its size, and the width of each entry. In addition, the analysis needs to determine the precise expression used to derive the target address from the jump table entries. This leads to four main sources of unsoundness and/or precision loss:

- *Limiting the length of backward slice:* For efficiency and to reduce precision loss, many techniques limit the length of the backward slice, e.g., angr [32] restricts it to three basic blocks [23].
- *Limiting the number of paths considered:* Another common approach is to limit the number of paths analyzed. For instance, Ghidra [30] assumes that jump table base and index can only be derived from a single path [23].
- *Pattern-driven analysis:* Given that jump tables are typically generated by compilers, patterns can often be identified in the computation of the target address. These are often leveraged in jump table analysis [1, 11, 18, 21, 37]. However, such an approach can make the analysis compiler-specific.
- *Limited capacity to analyze jump table bounds:* Underestimating the bound will lead to missed targets, while overestimation will degrade precision. As our results show, existing jump table analyses do not seem to pay sufficient attention to this problem.

To overcome these drawbacks, we present SJA (Static Jump Table Analysis), a new jump table analysis technique in this paper. In contrast with previous approaches, our technique is based purely on a forward analysis. Moreover, it considers all paths, and hence does not compromise on soundness. At the same time, we achieve performance that is better than most previous techniques. We also show that our analysis scales to large binaries, with an analysis rate of about 300 KB/s.

Our approach is based on abstract interpretation [6]. We introduce a new abstract domain that can accurately capture the computations used in jump tables, such as multiplying an index by a constant, adding an offset, dereferencing memory contents, etc. These computations are captured without relying on pattern matching. As a result, our approach is robust in the face of reordering or refactoring of operations. In contrast, such changes often cause pattern-matching to fail, leading to a loss of accuracy. We also present an effective and efficient bounds analysis for jump tables that leads to much better overall accuracy. Specifically, our approach reduces false negatives by 35× over previous techniques[1]. Based on F1-score, SJA's error rate is 3× lower than that of Dyninst, the leader in a recent comparison [23].

In summary, our main contributions are as follows:

- We devise a new abstract domain, together with appropriate abstract operations, to capture the "structure" of a code pointer. This approach avoids pattern-matching that is commonly used in many previous works. Pattern-matching can be reliant on

---

[1]A sound analysis should incur zero false negatives, but this applies only if the analysis is invoked on all code. On complex binaries, contemporary disassembly techniques can miss a small fraction of the code [23], with the result that our analysis is not even run on this code. This is the primary source of false negatives (i.e., missed jump table targets). In addition, we observe a handful of cases where our analysis reports ⊤ as the value of a code pointer — indicating that nothing is known about its value. In conventional static analyses, this would count as a false positive, with possible targets consisting of every instruction in the current function. However, previous works in jump table analysis report it as a false negative — all of them can discover the indirect jump instructions, but when they have no information about the destination, they report zero indirect targets [23], and hence treat it as an instance of false negatives. Thus, for consistency, we report these cases as false negatives as well.

```
switch(x) {
  case 0:
    return y+3;
  case 1:
    return 2*y;
  ...
  case 7:
    return y-4;
  default:
    return 0;
}
```

```
L1:  mov $6000,%r8
     mov $8000,%r9
     mov %r11,(%r10)
     cmp (%r10),$7
     ja  L3
L2:  mov %r11,%r12
     shl $2,%r12
     add %r9,%r12
     mov (%r12),%r13
     add %r8,%r13
     jmp *%r13
L3:  ...
```

```
L1:  R8 = 6000
     R9 = 8000
     *R10 = R11
     IF (*R10 >u 7)
       JMP L3
L2:  R12 = R11
     R12 = R12 ≪ 2
     R12 = R9 + R12
     R13 = *R12
     R13 = R8 + R13
     JMP *R13
L3:  ...
```

Fig. 1: A C/C++ switch statement, its assembly code, and the associated intermediate representation (IR)

compiler idioms and/or be sensitive to compilers or their versions. In contrast, a more systematic approach such as ours can cope better with syntactic changes that preserve semantics, such as reordering or refactoring of code pointer computation.

- We present a new bounds analysis technique that can handle assignments and comparisons equally effectively. Its key benefit is its ability to fully factor the bidirectional and transitive nature of equalities, whereas previous approaches tended to propagate constraints only in one direction, namely, from the right-hand side to the left-hand side of an assignment. We present techniques for representation and propagation of constraints that achieve efficiency together with increased power.
- We present a detailed experimental evaluation of our approach, comparing it with several existing systems, e.g., angr, Dyninst, Ghidra, and Ddisasm.
  - Our emphasis on soundness leads to a drastic 35× reduction in miss rates as compared to the best previous techniques.
  - Our false positive rate of 2% is better than all other systems except Dyninst. Moreover, when we compare on the error rate based on F1-score, our results are 3× better than the best among previous work.
  - Finally, we show that soundness is achieved without compromising on performance. SJA runs in $O(nm)$, where n is the number of instructions and m is the number of variables that have equality relationships with another. As m tends to be very small in practice, SJA's runtime performance is comparable and/or better than most previous systems.

2 Jump Table Overview and Approach Outline

A jump table consists of an array of code pointers/offsets used to compute the final target of an indirect jump. It results from the compilation of switch statements (or equivalent constructs) in high-level languages. Jump tables are also frequently used in hand-coded assembly. Fig. 1 illustrates a C/C++ switch statement and the jump table code resulting from it. This code is shown first in x86 assembly, and then in an intermediate language used in our analysis. This intermediate language is a register-based language and is similar to those used in compilers[2]. Jump table accesses fall into four major types:

- $*(TBase + Stride \times Index)$
- $Base + *(TBase + Stride \times Index)$

---

[2]The specific IR we use comes from the LISC [12] assembly-to-IR lifter that we use.
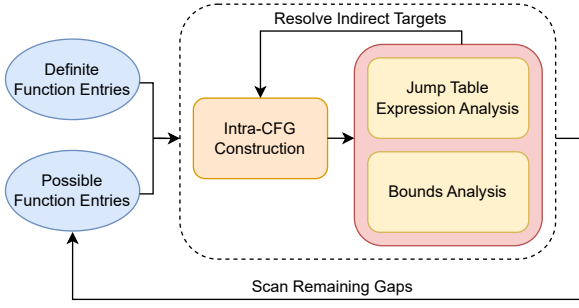
Fig. 2: SJA's Approach

- $Base + Stride \times Index$
- $*(TBase2 + Stride2 \times *(TBase1 + Stride1 \times Index))$

Here $TBase$ refers to the base of the jump table, which is an array that appears within read-only data, or in some cases, within code; and "$*$" denotes the memory dereferencing operator. In the first form, target code addresses are directly stored in the jump table. This is possible in position-dependent code. The second forms is most common in position-independent code, where a code offset is stored in the jump table. It is added to $Base$, a base address in the code region, in order to obtain the final jump target. Our illustrative example (Fig. 1) corresponds to this access type, with $TBase = 8000$ and $Base = 6000$. ($TBase$ and $Base$ are stored in registers $R8$ and $R9$ respectively.)

We have found a few instances of the third form in hand-written assembly. This form is possible only in the special case where code sizes are identical across all the cases in the jump table. The fourth form has been called a *nested jump table* in previous work [18, 37]. Specifically, it corresponds to a 2-level nested jump table where the first table stores a value that serves as an index in the second level table. More general cases are possible, i.e., n-level nesting for $n > 2$.

Most previous works have formulated jump table resolution as a backward analysis problem: starting from the indirect jump, these methods follow program control-flow backward to determine how the target was computed. Often, a pattern-matching approach is used. Many techniques also restrict the length of backward traversal in order to reduce the runtime. In contrast, we use a forward analysis for the following reasons:

- Soundness requires all paths to be analyzed. A purely backward approach cannot detect if there are incoming branches into a backward path being explored, since it can be an indirect transfer. So, a subsequent forward phase would be needed.
- A forward analysis approach is more amenable to a simple implementation in an abstract execution framework such as SJA. This, in turn, enables us to focus on designing the right abstract domain that can cope with the complexities of the jump tables mentioned above.

Fig. 2 illustrates our jump table analysis. First, we gather a set of *definite* function entries, e.g., program entry point, entries in the dynamic symbol table, and entries of default initialization and cleanup functions. These are fed into an analysis that discovers all of the function entry points in the binary. We rely on existing techniques, specifically function interface analysis [27], in this phase. Note that errors in function entry identification will cascade into errors in the output of jump table analysis. Importantly, when function discovery fails to report a legitimate function, that code is not analyzed

at all, which means that the jump tables present in such functions won't be identified. These false negatives will occur regardless of the soundness of analysis techniques used. (As mentioned before, this is the main source of false negatives in our method.)

For each function entry, we perform a forward intraprocedural analysis to resolve indirect jump targets. The analysis consists of two independent techniques: Jump table expression analysis and Bounds analysis. For jump table expression analysis, we present a new abstract domain that can handle all four jump table forms, including multi-level nested jump tables.

Bounds analysis recovers the size of jump tables using control-flow and dataflow information. Our novelty here is in terms of efficient support for handling equality relations. Specifically, even when bounds checks are missing on the index variable, our analysis can typically infer them from checks made on other variables whose values are related to the index variable.

As our jump table analysis discovers indirect targets and adds them to the CFG, new code becomes reachable. This new code needs to be analyzed for additional jump tables. In general, an n-level nested switch statement[3] will require n iterations of our analysis. While this may seem like a performance concern, such nesting is relatively rare and hence we have not optimized this case further.

In the following sections, we describe our jump table expression analysis (Sec. 3) and bounds analysis (Sec. 4). As these techniques rely on abstract interpretation, we include a short overview of this technique for the benefit of readers new to the topic.

Abstract interpretation [6]. One of the foundational techniques in program analysis that has formed the basis of most static binary analysis techniques [3, 7, 8, 27, 31, 32] is abstract interpretation. During normal execution, program variables take values over *concrete* domains such as integers. In abstract interpretation, programs are instead evaluated over *abstract domains.*

Abstract interpretation is frequently illustrated using the "rule of signs" example, where each numeric variable is abstracted into the domain $\{-ve, 0, +ve\}$. Here "$-ve$" stands for all $n < 0$ and $+ve$ for all $n > 0$, and "0" for the number zero. Thus, each point in the abstract domain corresponds to a subset of values in the concrete domain.

Based on this abstract domain definition, we can define the abstract equivalents $+_a$ and $*_a$ of the concrete operations $+$ and $*$. For brevity, we omit cases where one of the operands is zero.

x	y	$x +_a y$	$x *_a y$
$+ve$	$+ve$	$+ve$	$+ve$
$+ve$	$-ve$	$\top$	$-ve$
$-ve$	$+ve$	$\top$	$-ve$
$-ve$	$-ve$	$-ve$	$+ve$

Note how $+_a$ introduces approximations: when a positive and negative number are added, one cannot predict the sign of the output. (Multiplications don't introduce approximations in this abstract domain, but if we merged zero into $+ve$, this would no longer be true.)

Abstract interpretation can be understood as reevaluating a program using these abstract operations, while the variables themselves assume values from the abstract domain. At conditional branches,

---

[3]Note that a nested switch statement isn't the same as a nested jump table [18, 37]. With nested switch statements, each nesting level can have its own indirect jump statement. In contrast, a nested jump table uses a single indirect jump (but its target may be computed from multiple jump tables).

Domain $\mathcal{D} ::= \mathbf{D} \cup \{\top, \bot\}$
$\quad\quad\quad \mathbf{D} ::= B \mid B + S \times I \quad$ where $B, S \in \mathbb{Z},\ S \neq 0,\ I \in (\mathbf{V} \cup *\mathbf{D})$,
$\quad\quad\quad \mathbf{V}$ is the set of variables in the program

Expressions $e ::= c \mid v \mid e{+}e \mid e{-}e \mid e{\times}e \mid e{\ll}c \mid *e$
$\quad\quad\quad\quad (v$ is a variable, c is a constant$)$

Base cases

c	c
v	$0+1\times v$

Recursive cases

X	$B_1+S_1\times I$	$B_1+S_1\times I$
Y	$B_2+S_2\times I$	B_2
$X+Y$	$(B_1+B_2)+(S_1+S_2)\times I$	$(B_1+B_2)+S_1\times I$
$X-Y$	$(B_1-B_2)+(S_1-S_2)\times I$	$(B_1-B_2)+S_1\times I$
$X\times Y$	$\top$	$(B_1\times B_2)+(S_1\times B_2)\times I$
$X\ll Y$	$\top$	$(B_1\times 2^{B_2})+(S_1\times 2^{B_2})\times I$
$*Y$	$0+1\times *(B_2+S_2\times I)$	$0+1\times *B_2$

Fig. 3: Abstract domain for jump table analysis

there is typically not enough information to determine which side will be followed, so the typical approach is to follow both branches and then take the union of abstract values when control flows merge. By ensuring that abstract domains are *lattices*, unions (as well as intersections) of any two abstract values will yield another abstract value. This also ensures that there is a maximum element $\top$ (which corresponds to every possible concrete value) and a minimum element $\bot$ (which corresponds to the empty set).

An abstract interpretation is sound if the abstract value computed for every variable corresponds to a superset of the concrete values that this variable may have in any concrete execution. For instance, an abstract interpretation that assigns $\top$ to every variable is trivially sound (but not very useful).

3 Jump Table Expression Analysis

Accurate analysis of jump tables is necessary, or else control-flow graphs would be incomplete, which, in turn, can lead to unsoundness of all static analyses that use the CFG. For pragmatic reasons, there is a tendency in previous works to use pattern-matching on jump table analysis. The downside of this approach is that irregular patterns with similar semantics are not recognized. In fact, none of previous works systematically evaluate the effectiveness of their techniques. In this paper, we show that SJA can iteratively recognize many more jump tables using a principled static analysis.

Our abstract domain and its operations are shown in Fig. 3. The domain is designed to infer a formula that relates the values of (any) two variables in the program. In this regard, a variable corresponds to a definition of a register, i.e., multiple definitions of the same register will be treated as distinct variables.

We are not trying to capture all possible relations, but just those that arise in jump tables — specifically, the four major types discussed in the last section. For the third form, I corresponds to the index variable of the jump table. To support the other forms, we permit a recursive domain, where I is itself a formula. Multiple nesting levels enable analysis of multi-level nested jump tables.

L1 :	$R8=6000$	$R8=6000$	B-1
	$R9=8000$	$R9=8000$	B-1
	$*R10=R11$	$*R10=0+1\times R11$	B-2
	IF $(*R10 >_u 7)$	N/A	
	JMP L3	N/A	
L2 :	$R12=R11$	$R12=0+1\times R11$	B-2
	$R12=R12\ll 2$	$R12=0+4\times R11$	R-4.2
	$R12=R9+R12$	$R12=8000+4\times R11$	R-1.2
	$R13=*R12$	$R13=0+1\times *(8000+4\times R11)$	R-5.1
	$R13=R8+R13$	$R13=6000+1\times *(8000+4\times R11)$	R-1.2
	JMP $*R13$	N/A	
L3 :	. . .	N/A	

Fig. 4: Illustration of rules in Fig. 3 on our example

Recursive abstract domains have the potential to cause nontermination in the presence of loops. This happens because of fixpoint iteration. However, since we limit fixpoint iteration to terminate quickly (Sec. 5.4), recursion does not pose a serious problem.

The specific language subset relevant for this analysis is that of expressions e as shown in Fig. 3. The abstract interpretation for the base cases is shown first. It covers the cases of constants (c) and variables (v). Effectively, the abstraction of a constant or a variable is itself. The recursive cases are shown next: they correspond to the application of operators $+$, $-$, $\times$, $\ll$ and $*$ (memory dereferencing). The table does not show assignments because their semantics is independent of the domain. (See Sec. 5.3.1 for the details.)

The first two rows of recursive cases table (i.e., the two header rows) show different combinations of abstract values for the operands X and Y. The table body shows the result of applying each of the above-mentioned operators to X and Y. For instance, when $X = B_1 + S_1 \times I$ and $Y = B_2 + S_2 \times I$, their addition results in a point $(B_1 + B_2) + (S_1 + S_2) \times I$, as shown in the table. In other words, the bases are added, as are the strides. Note that this is applicable only when the index I matches. Otherwise, the result will be $\top$. To reduce clutter, Fig. 3 generally omits operand combinations that result in $\top$.

Fig. 4 illustrates these abstract operations on our running example from Fig. 1. The first column shows the instructions in their IR form, while the middle column shows the abstract value of the left-hand side operand after the execution of the instruction. The third column identifies the specific cell from Fig. 3 that is used to arrive at this result. Base case rules are referenced as B-1 and B-2. Recursive cases are referenced by a combination of row and column in the table body. For instance, R-1.1 refers to the cell containing $(B_1+B_2)+(S_1+S_2)\times I$.

At control-flow merge points, we need to apply the union operation in the domain. This can result in a loss of precision. To mitigate this, we extend the domain to $\mathcal{D}^*$, i.e., the abstract value can be a set of points in the domain $\mathcal{D}$ shown in Fig. 3. At merge points, we can define $\sqcup$ as the union of value sets from preceeding branches. In an implementation, the size of these sets can be capped at a small value, say, 10, as we have done in our implementation. If a set exceeds the size limit, its abstract value is set to $\top$.

Based on this analysis, jump table targets can be identified as follows. At an indirect jump, we retrieve the abstract value associated with the register that specifies the target. If it matches one of the forms introduced at the beginning of this section, we check if the table base and code base (if present) are within the read-only data or code region. Because jump table consists of contiguous entries, we iterate over possible *concrete* values of the index, retrieve the content
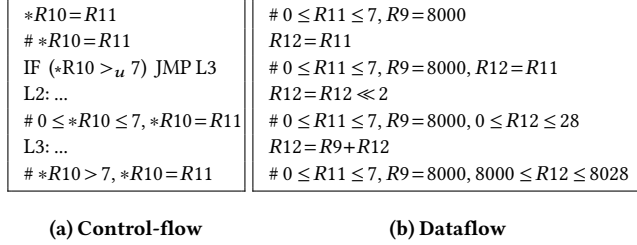
(a) Control-flow	(b) Dataflow

Fig. 5: Illustration of bounds analysis on our example

from the table, and compute the target. If the target is within the code region, we add an edge to the CFG.

Without information on possible bounds, this iteration must continue until we encounter a jump table entry that fails the above check. While this will help in terms of soundness, it can lead to considerable false positives. To mitigate this, we describe a bounds analysis in Sec. 4 that determines a range of concrete values of the index.

Advantages of abstract domain $\mathcal{D}$. VSA [3] is perhaps the most widely cited binary analysis technique based on abstract interpretation. It was proposed to compute an over-approximation of the set of *memory addresses*. Specifically, its abstract domain consists of points of the form $a + b \times [c, d]$, where a, b, c and d are integer constants. While VSA's domain has some similarity with SJA's, this is superficial: At its heart, jump table expression analysis is about the *relation* between an index variable I and the value of the jump table target. Meanwhile, VSA is a *value analysis* to compute the set of possible values possessed by the jump table target, but cannot relate it to the contents of the index variable.

One consequence of VSA's value-based approach is that it loses a lot of information on memory reads, often resulting in $\top$. For instance, consider the instruction $R13 = *R12$ in our working example. Assume that VSA had the accurate value of $R12$ as $8000 + 4 \times [0, 7]$. A sound approximation of the value of $R13$ would be the union of contents from memory locations $\{8000, 8004, ..., 8028\}$. Even in the optimistic scenario that the contents of all these locations is available statically, the union operation cannot, in general, yield a better result than $\top$. In contrast, our abstract domain $\mathcal{D}$ keeps operating on the relationships without reducing everything to a value, and hence do not suffer this loss.

4 Bounds Analysis

Bounds analysis gathers constraints on variable values (i.e., contents of memory and registers) that can be inferred on the basis of branch directions at conditional branches (control-flow), or arithmetic operations (dataflow), as illustrated in Fig. 5. Bounds analysis has broader uses, but in this paper, we primarily apply it for computing jump table sizes.

rev.ng [8] and OSRA [7] obtain constraints at conditional branches that follow comparisons with constants. These inequalities are propagated after assignments. However, their techniques do not reason about equality relationships between variable values and, as a result, fail to discover some implicit constraints. For instance, in both of the snippets shown below, the condition x < 5 will never hold. While these systems can infer this for the snippet on the left, they fail to do so for the snippet on the right.
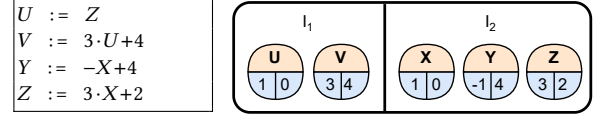
Fig. 6: Equality representation using *equivalence class*

```
if (y < 5) goto …          x = y
x = y                      if (y < 5) goto …
if (x < 5) …               if (x < 5) …
```

In contrast, we propose an efficient technique to compute equality relationships between variables (Sec. 4.1). In addition, we enhance this technique to incorporate concrete constraints obtained from relations between variables and constants (Sec. 4.2). Therefore, SJA can handle both the snippets shown above.

4.1 Efficient Handling of Equality Relations

Affine relations [13] was perhaps the earliest studied analysis to infer equality relationships between variable values. Affine relations are of the form $c_0 + c_1 X_1 + c_2 X_2 + \cdots + c_n X_n = 0$, where c_i are constants and X_i correspond to variable values. More recently, such abstractions have been called *symbolic abstractions* [9, 29]. The key problem here is to compute the (strongest) relation that holds at program location L, given the relations that hold at its predecessors. For affine relations, this step is expensive in practice [19], and hence the analysis does not scale. To avoid this complexity, we focus our analysis on the simpler case of equality relationships involving just two variables:

$$X_i = a \cdot X_j + b$$

As confirmed by our experimental results, this form is sufficient to handle the vast majority of cases that arise in jump tables. Variables in this regard correspond to registers and memory locations. In addition to memory locations whose addresses are known statically, we provide some support for indirect memory accesses (i.e., cases where the memory address is held in a register). Our memory model captures the semantics that whatever is stored in a location x will be returned when x is read back. (However, for soundness, if there is an intervening write to a location y, and no information is available to prove that $x \neq y$, then a subsequent read of x will return $\top$.)

There are two sources of equality relations in programs: equality comparisons and assignments. Of these two, assignments are the more frequent source, but we support both. Our goal is to capture the *symmetric* (also called *bidirectional*) nature of equality and allow constraint information to propagate in all directions.

Recall that we focus on relations of the form $X_i = a \cdot X_j + b$ where $a, b \in \mathbb{Q}$ and $a \neq 0$. Since equality is bidirectional, such X_i and X_j belong to the same *equivalence class* where the constraint on one variable can be inferred from any other variable in the same class. As a result, we can inject an extra variable, namely I_k, into a class and relate every variable in the class to I_k. Fig. 6 illustrates two such *equivalence classes*. The first class consists of variable U, V and an extra variable I_1. Specifically, line 2 can be rewritten as $U = 1 \cdot I_1 + 0$ and $V = 3 \cdot I_1 + 4$. The relationships between U, V and I_1 are illustrated in bubbles, each of which consists of a variable and its parameters. Note that this representation has many key advantages:
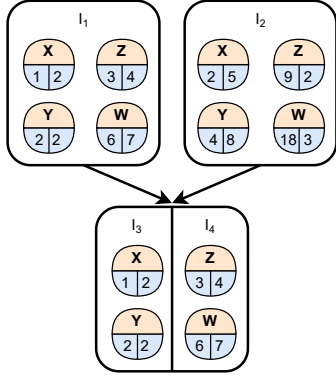
Fig. 7: An example of merging *equivalence classes*

Compact representation. Each variable can appear in *at most* one *equivalence class*. In addition, there is no linear relationship between variables in different classes.

Efficient handling of assignments. An important feature of this representation is that new assignments to a variable can be processed efficiently. For instance, Z initially belongs to the same class with U and V (line 1). However, after the assignment (line 4), the relationships between Z and U, V become invalid.[4]

To invalidate the relations that involve Z, we simply remove Z from the class where it belongs to. Moreover, if Z appears as a memory address in some class, this occurrence needs to be replaced by an equivalent expression on another variable that appears in the original class of Z. Finally, we insert Z into the appropriate class based on its new value, e.g., the second class with X, Y and an extra variable I_2.

Mitigating the impact of memory aliasing. Note that a memory location Z could be aliased to other memory locations. An assignment to Z invalidates not only the relations involving Z, but also those involving aliases of Z. Without additional information, it is possible that Z is aliased to *all* memory locations, and causes *all* relations involving memory to be discarded. This scenario could be detrimental to precision especially since assignments occur very frequently. However, it is feasible to rule out some memory aliasing for those addresses that can be expressed by *equivalence class*. In particular, suppose that R_i and R_j are registers in the same class, it is trivial to verify if the memory location $a \cdot R_i + b$ is not aliased to $c \cdot R_j + d$. However, if R_i and R_j belong to different classes, no linear relation between these memory addresses is captured. In such cases, for soundness, SJA assumes that these memory locations can be aliased to each other.

Efficient merge of paths. At branching statements, each branch starts with a separate copy of the relations from its predecessor and modifies its own relations while executing its instructions. At control-flow merge points, we typically compute a *disjunction* of *possible* equalities. However, since our goal is to find the relations that hold in *every* merging path, we instead compute a *conjunction* of *definite* equalities. These equalities can be represented by the system of *equivalence classes*.

Fig. 7 illustrates the merge of two branches. For simplicity, each branch has only one *equivalence class*. For instance, the relations $Y = 2 \cdot X - 2$ and $W = 2 \cdot Z - 1$ hold for both paths, and there is no common relation between X and Z in both paths. Therefore, the merging output consists of two *equivalence classes* associated to the extra variables I_3 and I_4. A typical approach to derive the output is to iterate over each pair of variables: if there is a common relation between them in both paths, these variables belong to the same *equivalence class*; otherwise, they belong to different classes. This approach is simple — the downside is that its time complexity is quadratic (over the number of variables) for each merge. In contrast, SJA merges two paths at linear time complexity for each merge by leveraging the relationships between extra variables, e.g., I_1 and I_2 in our examples.

Recall that X and Y belong to the same class if $\exists a,b \in \mathbb{Q} : Y = a \cdot X + b$.

$$\begin{cases} 2 \cdot I_1 + 2 = a \cdot (1 \cdot I_1 + 2) + b \\ 4 \cdot I_2 + 8 = a \cdot (2 \cdot I_2 + 5) + b \end{cases}$$

It can be seen that $Y = a \cdot X + b$ is satisfied if $2 \cdot I_1 + 2 = 4 \cdot I_2 + 8$ and $1 \cdot I_1 + 2 = 2 \cdot I_2 + 5$ hold. It can be inferred that X and Y are in the same class because these equations are equivalent. In fact, both of them can be reduced to the same relationship $I_1 = 2 \cdot I_2 + 3$. Meanwhile, the relationship between I_1 and I_2 is different for Z, specifically, $I_1 = 3 \cdot I_2 - \frac{2}{3}$. Consequently, X and Z do not belong to the same class.

The key advantage of our design is that we can compute the relationship between I_1 and I_2 for each individual variable, thereby achieving a linear time complexity for each merge. Variables with equivalent relationship between I_1 and I_2 are classified to the same *equivalence class*.

4.2 Embedding of Concrete Constraints

Most jump tables result from the translation of switch statements, whose labels are required to be constants in most languages, including C/C++. As jump tables consist of constant values known at compile time, their bounds are also constants. In the code, these bounds are typically expressed using comparisons with constants.

On x86-64 as well as AArch64 architectures, comparisons are decoupled from conditional branches: comparisons set flags, with branches depending only on these flag values. To identify the constraints associated with each branch direction, these two pieces of information need to be combined[5]. For instance, on x86-64, when the instruction flag = cmp(X, 3) is followed by a jle (jump if less or equal) branch, the constraint $X \leq 3$ is generated. SJA represents such concrete constraint in a uniform way using ranges, e.g., $X \leq 3$ can be formulated as $X \in [-\infty, 3]$.

A key feature with our approach is that concrete constraints can be embedded to the system of *equivalence classes* losslessly. Specifically, since all variables in same class can be derived from its corresponding I_k, SJA stores the appropriate range constraint in each I_k.

At merge points, we combine constraints from all predecessors using disjunction. We consider the example in Fig. 7. Suppose that $I_1 \in [-2, 5]$ and $I_2 \in [0, 2]$, and our target is to compute I_3. It can be derived that $X \in [0, 7]$ (left branch) and $X \in [5, 9]$ (right branch), respectively. Then, the constraint on I_3 in each branch can be inferred

---

[4]Note that if a variable is assigned to a function of itself, e.g., $Z = a \cdot Z + b$, it can still be inferred from other variables in the original class. In such cases, we do not invalidate its relations, but rather adjust the parameters and retain the variable in the class.

[5]In x86_64 and AArch64, many instructions can affect the flags but in practice, only a small subset of these instructions are involved in conditional branches, e.g., cmp, sub, test for x86. We find that this is true even in hand-coded assembly. Therefore, handling these instructions is sufficient to obtain jump table bounds.

by replacing X with $1 \cdot I_3 + 2$. In more detail, for the left branch, we have $0 \leq 1 \cdot I_3 + 2 \leq 7$, which is equivalent to $I_3 \in [-2,5]$. Similarly, $I_3 \in [3,7]$ can be derived for the right branch. Consequently, we can compute I_3 for the output class as follows:

$$I_3 \in [-2,5] \sqcup [3,7] \Leftrightarrow I_3 \in [-2,7]$$

At conditional branches, a new constraint is applied on top of the current constraint using conjunction. For instance, suppose that the above merging is followed by a branching statement with condition $X \geq 5$. As $X \geq 5$ can be translated to $I_3 \in [3,\infty]$, the new constraint is:

$$I_3 \in [-2,7] \sqcap [3,\infty] \Leftrightarrow I_3 \in [3,7]$$

5 Implementation

In this section, we describe our analysis framework design and implementation. The framework orchestrates and coordinates all of the steps involved in program analysis using abstract interpretation. It first lifts binary code into an architecture-neutral intermediate format, and then constructs a control-flow graph. It maintains the abstract store that captures the content of memory. Finally, it provides support fixpoint iteration, a technique needed to support loops and recursion. We describe each of these steps in more detail below. Our framework is modular and extensible, and provides an API for adding new abstract domains and analyses.

5.1 Lifting to IR

Similar to many previous binary analyses [4, 11, 18, 30, 32, 37], we first lift assembly instructions into an intermediate representation (IR) before analysis. This step abstracts most of the architecture specifics and complexities, thereby simplifying analysis implementation. We used the LISC system [12] for lifting assembly to GCC's IR called RTL. A key benefit of LISC is that it supports recent instruction set extensions, thus allowing us to handle almost any binary. We perform a one-time lifting for all instructions found by a disassembler of a target program.

5.2 Control Flow Graph (CFG) Construction

After lifting to IR, SJA constructs a control-flow graph for each "tentative" function. For stripped binaries, tentative functions start include (i) *definite function starts*, including the binary entry point, functions listed in the dynamic symbol table, and direct call targets; and (ii) *possible function starts,* such as relocated pointers that target a valid instruction, and "gaps" between other tentative functions that match function prologue signature.

At this point, indirect control flow targets are not resolved, so only directly reached basic blocks appear in CFGs. Once SJA resolves more indirect control flows (Sec. 3), we traverse from their indirect targets to construct more reachable basic blocks in the subsequent iterations. This process repeats until all code is identified and analyzed.

5.3 Abstract Store

Our "variables" can refer to either CPU registers or memory locations. The values of these variables are held in the *abstract store.* Similar to previous proposals such as VSA [3], our store is organized into multiple regions. In particular, our abstract store consists of *registers* and *stack*. Support for heap and global memory are very limited: the only semantics supported is that of ensuring that a read

of a memory location x produces the same value that was written. For soundness, we ensure that any intervening write to a location y invalidates this value that was written to x unless we can prove $x \neq y$. (More details on this handling of side-effects can be found in Sec. 5.3.3.) Our limited support for heap and global memory is useful in jump tables because the relevant code is often small in size and does not make too many memory accesses.

Register handling is largely straightforward — the only complication is that of the semantics of accessing 8, 16, 32, and 64 bit versions of the same register. This too is well understood, so the rest of this section is mainly focused on the abstract stack.

5.3.1 Accurate Modeling of Stack Memory. Stack memory is typically accessed using constant offsets from the value of SP at function entry. We leverage the *stack analysis* technique of Saxena et al. [31] in this regard. Specifically, their addresses have the value of $Base_{SP} + [c,c]$, where c is a constant that is derived by our analysis. Note that local variables are often accessed using an offset from the base pointer (BP) register. As most of the stack memory accesses refer to a single location in the abstract stack, this abstract domain can typically determine the (exact) value of BP as $Base_{SP} - [d,d]$ where d is another constant.

At function start, SJA initializes the content of each register R with the abstract value $Base_R$. Stack locations above the current SP can also be initialized this way — let us call them $Base_{P_1}, \ldots, Base_{P_n}$, the initial values of parameters 1 through n. Stack locations below the SP are initialized to $\top$. In addition, since the stack frame is just being allocated, valid references to these locations cannot appear anywhere else in memory or registers other than the SP. As a result, if a memory location is written using an address stored in memory or registers other than SP, it cannot target the current stack frame. This property is captured by our abstract store implementation, thus providing improved accuracy in reasoning about the stack contents. (This property is lost if a stack-derived address is subsequently stored in memory — we revisit this issue under the topic of "weak updates" below.)

5.3.2 Efficient Handling of Large Abstract State. It may seem that the memory needed for registers and the stack frame is relatively small. However, note that the abstract state changes with each instruction and that we need to be able to access the abstract state at any program location. As a result, it is necessary to replicate the abstract store after each instruction in a function. This replication is costly, so we take the following steps to increase efficiency.

First, we associate an abstract store with each basic block (BB) rather than every single instruction. When a BB is processed, its abstract state is updated after the abstract execution of each instruction in the BB. Second, each BB stores only the variables updated in that BB. This minimizes the size of the abstract store associated with that BB. For accessing a variable that wasn't assigned in a basic block B, we fetch its value from B's predecessors. We refer to this as *lazy loading*. If there are multiple predecessors, the values from the predecessor blocks are merged using the union operation in the domain. A lazy-loaded value is cached in B so that future accesses will be fast.

5.3.3 Weak Updates. A weak update occurs when static analysis fails to capture the exact memory address used in a write operation, e.g., we are updating a location $Base_{SP} + [l,h]$ with $l < h$. To handle this correctly, the entire range of memory locations l through h on

the stack will have to be updated. Moreover, since only one of these locations will be updated at runtime, and all the other locations will continue to have their old values, a weak update needs to store the union of the original and new values in each of these locations. The net effect is that weak updates can have a significant negative impact on analysis precision.

Weak updates arise due to three main reasons. First is an update of an element in an array that is allocated on the stack. In many cases, an analysis of the conditional branches preceding the access can yield bounds for the access, thereby limiting the scope of the update to a small region of the stack. A second reason is due to passing a pointer to a stack location to a callee. In this case, without an analysis of the callee, a static analysis would have to conservatively assume that the entire stack frame may be clobbered. This loss of information can be mitigated by analyzing the callee. A third case is when a stack location is stored in global or heap memory. When this happens, because we maintain no information about these memory regions, future updates involving any pointer stored in global memory can clobber the entire stack frame. To mitigate the impact of this worst-case outcome, we assume that if no stack-derived address below $Base_{SP}$ escapes to untracked memory (e.g., global or heap), the stack frame is preserved through calls. Moreover, we take the ABI as a specification and conclude that the callee-saved registers are also preserved through calls.

A second challenge posed by having weak updates is that it requires multiple memory locations to be updated after processing just a single instruction, and hence can impact the speed of static analysis. Based on our stack memory modeling, a write to memory address that is not derived from SP should not affect stack. However, a write to memory address that is too imprecise can trigger the entire stack frame to be clobbered. To bound this overhead, we maintain clobber operations in a separate "layer" that is superimposed on the main abstract store to derive the ultimate content of memory. This allows clobbers to be recorded in constant time, at a slight increase to the cost of all read operations on the abstract stack (due to the need to check the clobber layer).

5.4 Fixpoint Iteration

To handle recursion and loops, *fixpoint iteration* is used. This is an iterative equation solving technique that begins with $\perp$ as the initial approximation for all variables. The $n + 1$th approximation is obtained by starting with the results of nth approximation. When the abstract domains are finite (or more accurately, when there are no infinite ascending chains), this iterative process will terminate. However, it may take time which is exponential in the size of the program. For infinite domains, it may not terminate at all. One way to avoid this complexity explosion is to begin fixpoint iteration at $\top$ instead of $\perp$. Such an approach would compute the *greatest fixpoint* (i.e., the largest solution). The key benefit of starting with $\top$ is that the results are sound even before a fixpoint is reached, so we can stop after k iterations for some small constant k. As a result, we can limit the overall analysis complexity to be linear, which is important for scalability.

Although starting from $\top$ entails some precision loss, accurate handling of recursion is not essential for jump table analysis. This is because jump table is generated by compilers statically, and thus does not depend on runtime values computed inside loops. Consequently, SJA defaults to starting fixpoint iteration from $\top$ unless configured otherwise for a specific analysis.

6 Evaluation

Our experiments evaluate SJA in terms of accuracy and performance. We compare SJA with 4 other tools: Dyninst, angr, Ghidra and Ddisasm. We did not include rev.ng in the evaluation because we were unable to get it to compile and run.

6.1 Experimental Setup

Evaluation platforms. All experiments were carried out on a desktop running Ubuntu 20.04 on 12th generation Intel Core i7 processor with 16 GB main memory and 200 GB SSD.

Benchmarks. Our framework was evaluated with Pang et al.'s benchmarks [23], which contain programs and libraries written in C/C++ compiled using GCC-8.1.0 and LLVM-6.0.0 on 6 optimization levels. Since SJA's implementation is for x86_64 on Linux, we used only the x86_64 ELF binaries in our evaluation.

Ground truth. Pang et al. extends the approach in CCR [15] to customize both LLVM and GCC to generate jump table ground truth, which includes details about jump tables (e.g., base, size and stride) accessed by each indirect jump. We also reuse the scripts in this work to generate control-flow graph constructed by angr, Dyninst and Ghidra. Regarding Ddisasm, we customize an official example `cfg-paths.py` to emit control-flow graph. Our custom python script is very simple and contains only 7 LoCs.

6.2 Accuracy

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

- How accurate are CFGs? How many missing edges and spurious edges are being added based on jump table analysis? (Sec. 6.2.1)
- Suppose there are n jump tables, what fraction of them are identified by jump table analysis? (Sec. 6.2.2)

6.2.1 Control-Flow Graph Accuracy. Jump table is generally a low-level construct generated by compilers, so its size can be extracted using the approach by CCR [15] and Pang et al. [23]. Note that multiple indirect jumps can access the same jump table, with each indirect jump referencing a different subset of entries in this table. However, the ground truth does not provide these details on a per-indirect-jump basis. Instead, it (a) lists the properties of jump tables, and (b) associates a jump table to each indirect jump. Since all analysis tools report CFG edges separately for each indirect jump, in order to make use of the available ground truth, we combine the indirect targets reported by analysis tools across all uses of the same jump table, and compare this union against the ground truth. We then evaluate CFG missing edges and spurious edges through the jump table entries identified by each analysis tool.

Specifically, for each jump table, we compare the set J_T of edges reported by an analysis tool against the set of valid edges J_G reported in the ground truth for that table. $J_G - J_T$ denotes the set of edges unrecognized by the tool, and hence are false negatives (FNs). Similarly, $J_T - J_G$ is the set of spurious edges reported by a tool that are not present in the ground truth, so they are false positives (FPs). [6] Precision, Recall and F1-score can be derived from FPs, FNs and TPs

---

[6]Pang et al. count false negatives only if the tool reports zero targets at an indirect jump. This means that a tool that reports only incorrect targets will not incur false negatives. It is also unclear if their false positives considers the number of spurious jump targets reported, but may instead be scoring the indirect jump itself. As a result, the numbers they report are somewhat different from ours. By counting the number of correct targets, our results are a more accurate reflection of analysis accuracy.
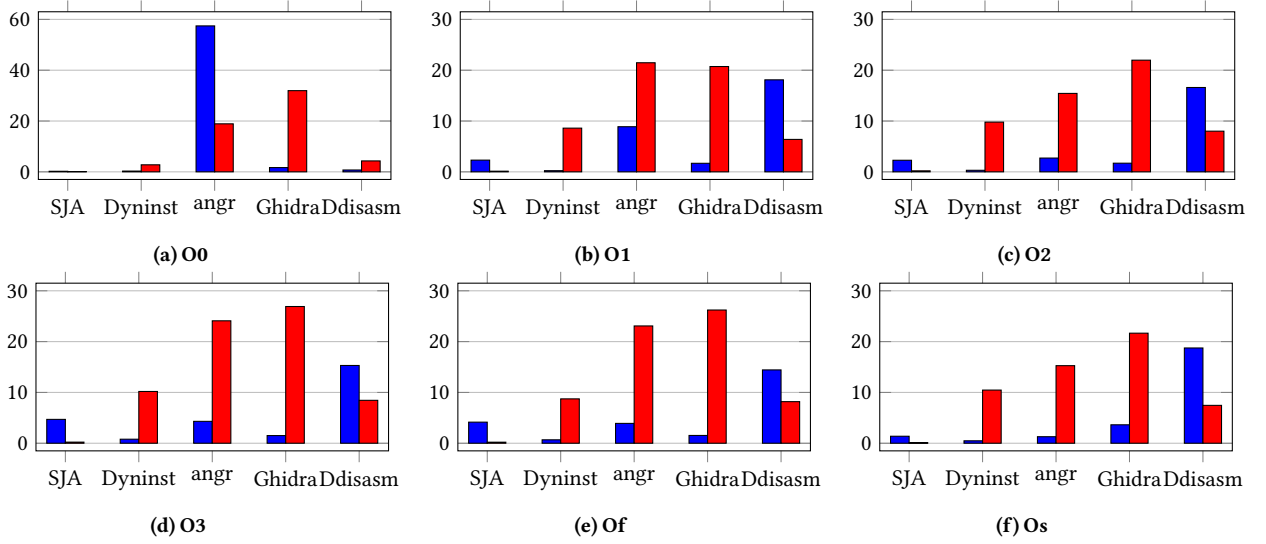
Fig. 8: Error rates in recognizing jump table entries. Blue and red bars are (1-Precision) and (1-Recall), respectively (%)

Table 1: Accuracy of jump table entries identification (%).

	Dyninst	angr	Ghidra	Ddisasm	SJA
Precision	99.5	78.4	98.1	85.9	97.4
Recall	91.6	79.7	74.7	92.7	99.8
F1-score	95.4	79.0	84.8	89.2	98.6

(as J_G in the ground truth). These measurements reflect the overall accuracy of our overall techniques (Sec. 3 and Sec. 4) as compared to other tools.

Note that missing edges, which affect the soundness of analysis techniques, are reflected in recall rate. Tab. 1 shows that SJA achieves a recall rate of 99.8%, which is translated to a miss rate of only 0.2% of jump table entries. Compared to previous techniques, SJA's miss rate is at least $\approx 35\times$ lower than others. Among SJA's competitors, Ddisasm is comparable with Dyninst at $\approx 7\%$ to 8% miss rate, while Ghidra misses more jump table entries than others, roughly 25.3% on average. In addition, Fig. 8 shows that Ghidra is also inconsistent across different optimization levels, e.g., it reaches about 20% miss rate at O2 optimization, but rises up to 32% miss rate at no optimization.

In terms of spurious edges, Tab. 1 shows that SJA achieves a precision rate of 97.4%. Compared to SJA, Dyninst and Ghidra seem to reflect a somewhat biased choice with high precision in exchange of low recall. For instance, Dyninst achieves 99.5% precision rate, but suffers from a relatively low recall rate of just above 90%. On the other hand, despite being slightly better than Dyninst at recall rate, Ddisasm achieves a much lower precision rate, at 86%. Moreover, Fig. 8 shows that SJA's error rates are stable, e.g., they fall between 0% to 4% across different optimization levels. Meanwhile, angr has a very poor precision rate of under 60% with binaries without optimization (O0), which brings its overall precision rate down to 78%.

It is important to note that the reported precisions above reflect our bounds analysis (Sec. 4). Without bounds information, the jump table size can be overapproximated using the adjacency heuristics [37], which assumes that jump table entries are placed sequentially. In other words, we expand a jump table as long as all consecutive

entries target valid instructions with respect to the provided disassembly. However, when applying this heuristics blindly, we found a very poor precision rate of 20%. Unlike general cases where unification often causes imprecision due to abstraction/relaxation of the actual constraints implied by the code, our bounds analysis captures true effect of satisfied (or unsatisfied) branches, so it doesn't lead to additional imprecision. As a result, our bounds analysis can effectively improve precision, e.g., increase from 20% to 98%, without sacrificing our high recall rate.

In terms of F1-score, SJA achieves the highest score of 98.6%. Compared to the other tools, SJA's error rate is 3× to 15× lower. Among the competitors, Dyninst achieves the highest F1-score of above 95% due to its high precision rate. Meanwhile, angr receives the lowest score of under 80%, where both precision and recall are equally low.

6.2.2 Jump Table Accuracy. Suppose that there are a few very large jump tables and many small jump tables, correct identification of the large ones is sufficient to achieve good CFG accuracy. However, this doesn't truly reflect the ability to identifying jump tables. Therefore, we compare analysis tools in terms of the number of jump tables correctly identified.

Since most tools don't record details of a jump table, direct comparison against ground truth is not possible. Therefore, we present a metric to evaluate the technique indirectly. Recall that J_G and J_T are the sets of targets reported in the ground truth and by an analysis tool, respectively. When it fails to resolve an indirect jump, it can either say (i) any address is a possible targets (FPs), or (ii) there is no targets (FNs). Based on this observation, we derive a new metric. Specifically, an analysis tool is considered to be able to identify a jump table if (a) J_T contains at least $a \times J_G$ valid targets, and (b) the number of entries reported is constrained below $b \times |S_{gt}|$. In other words, we select thresholds (a, b) and report the number of jump table satisfy the mentioned conditions. To avoid biases against FPs and FNs, we choose multiple different thresholds, specifically, $(a = 50\%, b = 200\%)$ and $(a = 90\%, b = 110\%)$, to show the advantanges of our techniques over the existing tools.
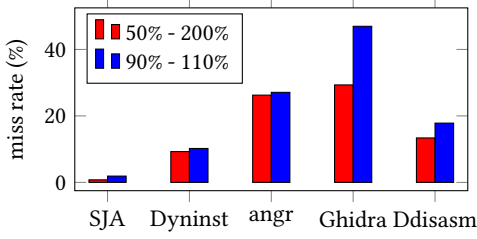
Fig. 9: Jump tables misses with different thresholds

Fig. 9 illustrates the miss rate of different tools in detecting jump table bases across different optimizations. It shows that SJA has consistently low miss rates from 0.8% to 1.9% across different thresholds. In comparison to other tools, we are up to 12× lower miss rates. Among other tools, only Dyninst misses about 10% of jump table bases, while Ddisasm is second from 13% to 18%. Meanwhile, Ghidra and angr miss more than a quarter of all jump table bases. Moreover, Ghidra is very sensitive to the thresholds, and fluctuates from 30% to 47%.

Additionally, we observed that *all* the tools were affected by increasing the lower threshold (50%-90%) and not the upper threshold (200%-110%). This tells us that current systems tend to minimize FP while sacrificing the FN rate[7]. In comparison, SJA achieves superior FN rate while maintaining a comparable FP rate.

6.3   Ablation Evaluation

In this section, we evaluate the key aspects of our approach in Sec. 3 and Sec. 4 and compare with existing techniques below:

- *VSA's abstract domain*: used in tracking jump table expressions by angr and Dyninst [23]. We evaluate the effectiveness of our abstract domain $\mathcal{D}$ (Sec. 3) in improving recall rate (See Sec. 6.3.1).
- *Unidirectional constraints*: only propagate constraints in one direction, that is from the right-hand side to the left-hand side of an assignment, i.e., it doesn't reason about equality relations in general cases. We measure the effectiveness of bounds analysis (Sec. 4) in improving precision rate (See Sec. 6.3.2).

6.3.1   Abstract Domain $\mathcal{D}$. Tab. 2 breaks down the effectiveness of our abstract domain (Sec. 3). As previously explained, VSA's domain is a value-based analysis, and hence it loses a lot of information on memory reads, which is usually a part of jump table expressions. As a result, analysis tools such as Dyninst and angr have to compensate the limitation of VSA with pattern matching and/or ad hoc rules [23].

Instead of incorporating heuristics, we reimplement VSA's abstract domain, which is capable of tracking regular *memory addresses*. Specifically, this domain is expressive enough to track the 1st and 3rd type of jump table access in Sec. 2. We then combine VSA's abstract domain with existing bounds technique. Our experiment shows that this approach misses about 21.4% of jump table entries. Specifically, we notice that VSA misses *all* the jump tables in libc-2.27.so compiled by gcc in O2 optimization since it only contains the jump tables of 2nd type in Sec. 2.

In contrast, SJA doesn't rely on incomplete patterns or limit the scope analysis: it only misses about 0.2% as $\mathcal{D}$, which is about 100×

---

[7]Some of these tools implemented ad hoc rules that exclude jump tables whose size is beyond certain limit [23].

Table 2: Accuracy of our techniques and previous techniques

	VSA's domain Unidirectional	**Domain $\mathcal{D}$** Unidirectional	**Domain $\mathcal{D}$** **Bidirectional**
Precision	80.4	79.9	97.4
Recall	78.6	99.8	99.8

lower than the miss rates using VSA's domain. This highlights the significance of our abstract domain $\mathcal{D}$ in improving recall rate.

6.3.2   Bidirectional Bounds Analysis. Similar to some previous works [7, 8, 11, 18, 37], *Unidirectional constraints* supports control-flow and dataflow inequality constraints, and also supports unidirectional propagation of constraints through assignments, e.g., from right-hand side to left-hand side [7, 8]. Meanwhile, *Bidirectional constraints* supports inferences from equality relations in both directions of assignments as well as equality comparisons. Note that for both bounds analysis techniques, if certain bounds cannot be derived, we fall back on adhering the adjacency rule [37] for soundness at the cost of low precision. Despite that, our experiments highlight that full support for equality relations doesn't necessarily mean significant precision loss. In fact, bidirectional propagation of constraints achieves 97.4% precision rate, as compared to just 80% as in unidirectional propagation of constraints, i.e., the error reduction rate is 8×. Unlike Dyninst and Ghidra, SJA can achieve this level of precision while still maintaining a miss rate that is at least 35× lower than that by any existing tool.

6.4   Scalability

In this section, we aim to break down the evaluation of SJA's runtime performance into two different aspects:

- How scalable SJA is in comparison with the other tools? (Sec. 6.4.1)
- An in-depth examination of the runtime performance of the static analysis engine (Sec. 6.4.2)

6.4.1   Evaluation of Total Runtime. To get an idea if SJA's complex analysis will scale to large binaries, we measure the total runtime performance of jump table identification with respect to code size and compare it with that of other contemporary works. Fig. 10 summarizes the result. Except for Ghidra, the remaining tools appear to scale linearly with respect to binary size at different slopes. SJA also exhibits a linear relation with binary size, e.g., SJA is able to analyze binaries of size close to 36 MB in approximately 3 minutes.

We break down the performance into different code size range. Note that Ddisasm performs better than SJA for binaries of size less than 100 KB. Fig.10 shows that SJA's runtime performance is constant (≈ 2 seconds) across these group of binaries. This is mainly because SJA's binary lifter consumes this constant time, regardless the amount of code to be lifted. While SJA performance continues to be linear in terms of binary size, Ddisasm's analysis time increases significantly for binaries of size greater than 100 KB, partly because their relational analysis that maintains a number of relations quadratic to the number of < *Register,Location* >. In addition, a similar trend is seen for angr as well. For smaller binaries of size less than 30 KB, angr's analysis time is close to that of SJA. However, it increases steeply and takes around 10× more time for binaries of size greater than 10 MB. Furthermore, Ghidra shows a consistently higher analysis time (≈ 20×) than SJA for binaries of all sizes. This
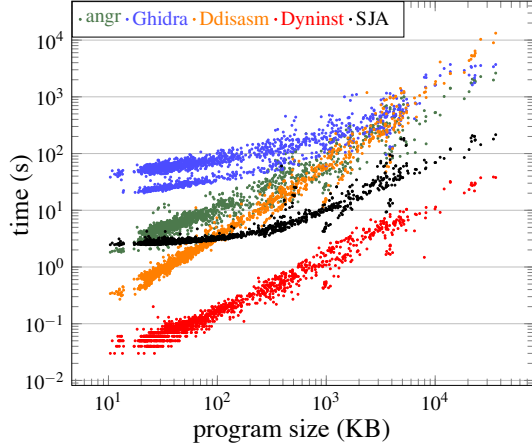
Fig. 10: Runtime performance on Pang et al.'s benchmarks

Fig. 11: Scalability of analysis performance per function

shows that state-of-the-art tools such as Ddisasm do not scale well with binary size. On the other hand SJA manages to analyze large binaries in a reasonable time. Dyninst performs better (6× less time), however its error rates are also much higher than that of SJA.

6.4.2 Evaluation of Analysis Time. We previously report the total runtime performance of SJA in comparison with the other tools. SJA consists of a few smaller steps, e.g., (a) disassembly, (b) binary lifting, (c) CFG construction, and (d) analysis time. Among them, the analysis task usually stands out as a major contributor to runtime performance. For instance, Ddisasm incorporates an expensive relational analysis as previously mentioned. Moreover, scalability is one of the reasons why existing tools do not analyze all paths, e.g., angr and Dyninst restricts the backward slicing scope for the same reason [23]. Therefore, we break down the performance efficiency of the analysis step. Since our analysis is intraprocedural, we report the analysis performance with respect to function size.

Recall that in bounds analysis, the complexity of a merge or a branching is $O(m)$, where m denotes the number of variables that have equality relationship with another. As a result, the complexity of analyzing a function is $O(nm)$, where n is the function size. Despite that, Fig. 11 shows that SJA's analysis time is roughly linear in terms of function size. In fact, SJA can analyze code at the rate of ≈ 300 KB/s. Perhaps, SJA is scalable because m tends to be small in practice. For instance, memory aliasing can invalidate the relations involving a lot of memory locations. Meanwhile, there are just dozens of registers to keep track with.

Fig. 11 shows two linear segments. For functions below 3K bytes, the slope is less than that for larger functions. We believe this difference comes from cache performance. With smaller functions, our analysis uses less memory and hence can more easily fit in the caches. As the function size increase, more of the accesses go to L3 or main memory, and this increases the slope of the line. Nevertheless, the overall trend is linear for larger function sizes as well.

7 Related Work

Accurate jump tables are essential in disassembly and control-flow graph construction, which provide the basis for binary instrumentation and binary analysis. As a result, jump table has been studied
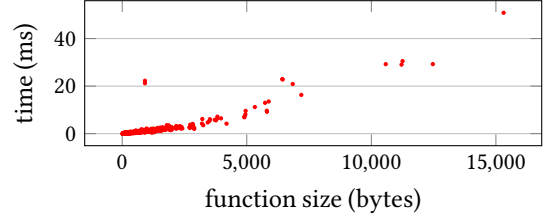
extensively by numerous existing works [1, 2, 4, 5, 8, 10, 11, 14, 16, 18, 21, 23, 24, 26–28, 30, 32–35, 37, 39–41, 41].

Pang et al. [23] quantitatively evaluate the disassembly performance of numerous tools, including jump table accuracy. However, since jump table accuracy is one of the many factors considered, they do not provide detailed justification for the metrics they used. By performing a more rigorous evaluation, we show that state-of-the-art tools misidentify at least 10% of jump tables.

Most existing works use backward slicing to resolve indirect jump targets. To improve scalability and/or precision, they tend to analyze only a part of the control-flow graph, and thus their techniques are unsound. In particular, angr [32] and Dyninst [11, 18] limit the scope to a few basic blocks (or a few instructions) by default [23]. Ghidra [30] assumes that both jump table base and index can be detected in a single path [23]. Egalito [37], Jima [1] and BOLT [21] use syntactical pattern-based techniques to identify jump tables, so their results are more platform-specific. In contrast, SJA analyzes all paths using forward analysis and abstract interpretation. Note that our approach can also detect nested jump tables that have been reported by previous tools (e.g., Dyninst and Egalito).

rev.ng [8], OSRA [7] use forward analysis to recognize a simple form of memory address that captures parts of jump table expressions. However, they either lack of discussion or implement ad hoc rules to detect jump tables. In addition, rev.ng and OSRA does not support equality relations, thereby unable to infer the size of jump table in complex cases. On the other hand, Ddisasm [10] derives jump table expressions from relations between registers. That said, their technique fails to capture expressions beyond registers, such as nested jump tables. Moreover, instead of handling of constraints, it implements ad hoc rules to determine the size of jump table. The limited inference capabilities of these approaches result in some bounds going undetected, which increases FPs. In contrast, SJA addresses this challenge by using an abstract domain that can represent equality relations effectively. As a result, our technique can reduce FPs from over 400% to just around 2%.

8 Conclusions

In this paper, we presented a new jump table analysis technique that achieves soundness without compromising on performance and scalability. It is based on a new abstract domain that we designed to capture the computation used to derive a code pointer. We also presented a new and efficient bounds analysis technique for determining jump table bounds. By combining these techniques, we achieve greatly improved false negatives — just 0.2% — while incurring a low false positive rate of 2%. Compared on the error rates based on F1-score, our results represent a 3× improvement over the best results reported in the literature.

9 Data-Availability Statement

The source code of SJA and relevant scripts to reproduce of the results are available [20]. The dataset and groundtruths, which are based on a prior work, are also published [22].

References

[1] Jim Alves-Foss and Jia Song. 2019. Function boundary detection in stripped binaries. In *ACSAC*. https://doi.org/10.1145/3359789.3359825

[2] Dennis Andriesse, Asia Slowinska, and Herbert Bos. 2017. Compiler-agnostic function detection in binaries. In *IEEE S&P*. https://doi.org/10.1109/EuroSP.2017.11

[3] G. Balakrishnan and T. Reps. 2004. Analyzing memory accesses in x86 executables. In *Compiler Construction*. https://doi.org/10.1007/978-3-540-24723-4_2

[4] D. Brumley, I. Jager, T. Avgerinos, and E. Schwartz. 2011. Bap: a binary analysis platform. In *CAV*. https://doi.org/10.1007/978-3-642-22110-1_37

[5] Sanchuan Chen, Zhiqiang Lin, and Yinqian Zhang. 2021. SelectiveTaint: Efficient Data Flow Tracking With Static Binary Rewriting. In *USENIX Security*.

[6] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Principles of programming languages*. https://doi.org/10.1145/512950.512973

[7] Alessandro Di Federico and Giovanni Agosta. 2016. A jump-target identification method for multi-architecture static binary translation. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. https://doi.org/10.1145/2968455.2968514

[8] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. rev.ng: a unified binary analysis framework to recover CFGs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*. https://doi.org/10.1145/3033019.3033028

[9] Matt Elder, Junghee Lim, Tushar Sharma, Tycho Andersen, and Thomas Reps. 2014. Abstract domains of affine relations. *ACM TOPLAS* (2014). https://doi.org/10.1145/2651361

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

[11] Laune C Harris and Barton P Miller. 2005. Practical analysis of stripped binary code. *ACM SIGARCH* (2005). https://doi.org/10.1145/1127577.1127590

[12] Niranjan Hasabnis and R Sekar. 2016. Lifting assembly to intermediate representation: A novel approach leveraging compilers. In *ASPLOS*. https://doi.org/10.1145/2872362.2872380

[13] Michael Karr. 1976. Affine relationships among variables of a program. *Acta Informatica* (1976). https://doi.org/10.1007/BF00268497

[14] Sun Hyoung Kim, Cong Sun, Dongrui Zeng, and Gang Tan. 2021. Refining Indirect Call Targets at the Binary Level. In *NDSS*. https://doi.org/10.14722/ndss.2021.24386

[15] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P Kemerlis, and Michalis Polychronakis. 2018. Compiler-assisted code randomization. In *Security and Privacy*. https://doi.org/10.1109/SP.2018.00029

[16] Draper Laboratory. 2022. *CBAT: A Comparative Binary Analysis Tool*. Technical Report. ONR TPCP.

[17] Fan Long, Stelios Sidiroglou-Douskos, and Martin Rinard. 2014. Automatic runtime error repair and containment via recovery shepherding. *PLDI* (2014). https://doi.org/10.1145/2666356.2594337

[18] Xiaozhu Meng and Barton P Miller. 2016. Binary code is not easy. In *ISSTA*. https://doi.org/10.1145/2931037.2931047

[19] Markus Müller-Olm and Helmut Seidl. 2004. A note on Karr's algorithm. In *International Colloquium on Automata, Languages, and Programming*. https://doi.org/10.1007/978-3-540-27836-8_85

[20] Huan Nguyen. [n. d.]. Static Jump Table Analysis. https://doi.org/10.5281/zenodo.12670597

[21] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. Bolt: a practical binary optimizer for data centers and beyond. In *IEEE/ACM CGO*. https://doi.org/10.1109/CGO.2019.8661201

[22] Chengbin Pang. [n. d.]. Dataset. https://github.com/junxzm1990/x86-sok.

[23] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. 2021. SoK: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *IEEE S&P*. https://doi.org/10.1109/SP40001.2021.00012

[24] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. 2012. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Security and Privacy*. https://doi.org/10.1109/SP.2012.41

[25] Soumyakant Priyadarshan, Huan Nguyen, Rohit Chouhan, and R Sekar. 2023. SAFER: Efficient and Error-Tolerant Binary Instrumentation. In *USENIX Security*.

[26] Soumyakant Priyadarshan, Huan Nguyen, and R. Sekar. 2020. Practical Fine-Grained Binary Code Randomization. In *ACSAC*. https://doi.org/10.1145/3427228.3427292

[27] Rui Qiao and R Sekar. 2017. A Principled Approach for Function Recognition in COTS Binaries. In *Dependable Systems and Networks (DSN)*. https://doi.org/10.1109/DSN.2017.29

[28] Nilo Redini, Ruoyu Wang, Aravind Machiry, Yan Shoshitaishvili, Giovanni Vigna, and Christopher Kruegel. 2019. BinTrimmer: Towards static binary debloating through abstract interpretation. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, (DIMVA 2019)*. https://doi.org/10.1007/978-3-030-22038-9_23

[29] Thomas Reps, Mooly Sagiv, and Greta Yorsh. 2004. Symbolic implementation of the best transformer. In *VMCAI*. https://doi.org/10.1007/978-3-540-24622-0_21

[30] Roman Rohleder. 2019. Hands-on Ghidra - A Tutorial about the Software Reverse Engineering Framework. In *Proceedings of the 3rd ACM Workshop on Software Protection*. https://doi.org/10.1145/3338503.3357725

[31] Prateek Saxena, R Sekar, and Varun Puranik. 2008. Efficient fine-grained binary instrumentation with applications to taint-tracking. In *CGO*. https://doi.org/10.1145/1356058.1356069

[32] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. 2016. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *Security and Privacy (SP)*. https://doi.org/10.1109/SP.2016.17

[33] Victor Van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. 2015. Practical context-sensitive CFI. In *ACM CCS*. https://doi.org/10.1145/2810103.2813673

[34] Victor Van Der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *Security and Privacy (SP)*. https://doi.org/10.1109/SP.2016.60

[35] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making Reassembly Great Again. In *NDSS*. https://doi.org/10.14722/ndss.2017.23225

[36] Mark Weiser. 1984. Program slicing. *IEEE TSE* (1984). https://doi.org/10.1109/TSE.1984.5010248

[37] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P Kemerlis. 2020. Egalito: Layout-Agnostic Binary Recompilation. In *ASPLOS*. https://doi.org/10.1145/3373376.3378470

[38] Miuyin Yong Wong, Matthew Landen, Manos Antonakakis, Douglas M Blough, Elissa M Redmiles, and Mustaque Ahamad. 2021. An inside look into the practice of malware analysis. In *ACM CCS*. https://doi.org/10.1145/3460120.3484759

[39] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical control flow integrity and randomization for binary executables. In *IEEE S&P*. https://doi.org/10.1109/SP.2013.44

[40] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R Sekar. 2014. A platform for secure static binary instrumentation. *ACM VEE* (2014). https://doi.org/10.1145/2576195.2576208

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