JTR: A binary solution for switch-case recovery

Lucian Cojocar, Taddeus Kroes, and Herbert Bos

Vrije Universiteit Amsterdam
lucian.cojocar@vu.nl, t.kroes@vu.nl, herbertb@cs.vu.nl

Abstract. Most security solutions that rely on binary rewriting assume a clean separation between code and data. Unfortunately, jump tables violate this assumption. In particular, switch statements in binary code often appear as indirect jumps with jump tables that interleave with executable code—especially on ARM architectures. Most existing rewriters and disassemblers handle jump tables in a crude manner, by means of pattern matching. However, any deviation from the pattern (e.g., slightly different instructions) leads to a mismatch.
Instead, we propose a complementary approach to "solve" jump tables and automatically find the right target addresses of the indirect jump by means of a tailored Value Set Analysis (VSA). Our approach is generic and applies to binary code without any need for source, debug symbols, or compiler generated patterns. We benchmark our technique on a large corpus of ARM binaries, including malware and firmware. For `gcc` binaries, our results approach those of IDA Pro when IDA has symbols (which is generally not the case), while for `clang` binaries we outperform IDA Pro with debug symbols by orders of magnitude: IDA finds 11 of 828 switch statements implemented as jump tables in SPEC, while we find 763.

1 Introduction

Solving indirect control flow transfers such as jump tables in a disassembler is important for many applications—from binary rewriting to reverse engineering, and from malware analysis to code complexity metrics [47, 37, 21]—because it is essential to find some parts of the Control Flow Graph (CFG) of a program. Unfortunately, it is also very difficult and modern disassemblers frequently get it wrong in cases where code does not follow common, easy-to-fingerprint patterns, such as handwritten assembly or malware.

Extracting a reliable CFG requires the ability to *distinguish* between data and code and to *solve* the indirect control transfers—in the sense of finding the possible targets for such transfers. Any over-approximation adds spurious edges to the CFG, while under-approximations remove legitimate edges.

Unless they can extract the CFG reliably, many binary analysis techniques either no longer work at all, or with reduced accuracy. Besides reverse engineering in general, this includes the analysis of code complexity [47, 37, 21] and binary control flow testing [6, 58]. Moreover, a reliable solution for jump tables also serves to detect the presence of custom protocol parsers [21].

A version of this manuscript was published at ESSoS'17. DOI: https://doi.org/10.1007/978-3-319-62105-0_12

If incorrect CFGs are a nuisance for software testers and reverse engineers, they can be downright catastrophic for binary rewriting solutions. Many software hardening approaches rely on binary rewriting [42] to offer security guarantees. Examples include control flow integrity (CFI) [26, 50, 6, 58, 52, 22], sandboxing [38, 18, 36, 55, 27, 25, 45, 57], static taint tracking [9, 19, 54, 28, 41]. An incomplete or incorrect CFG can void the security guarantees or even break legitimate software. Most binary rewriting solutions [12, 8, 49, 42, 51] are conservative when the CFG is incomplete, trading security guarantees for the overhead of the binary solution.

State-of-the-art disassemblers use pattern matching to solve complicated indirect control flow transfers. For instance, if a specific compiler generates a jump table to implement a switch statement in C, IDA Pro should know the precise template that the compiler will use *a priori*, so that it can search for exactly this pattern in the binary. Getting it right is important, as IDA uses the resulting jump targets to continue disassembly. Changing the code, however slightly, to not fit the template, results in a misclassification of the code. In practice, we found such cases in both benign and malicious software.

In this paper, we present a generic technique to solve indirect control transfers without pattern matching, to handle complicated cases—malware and handwritten code—for which templates are not available. We do not necessarily aim to outperform solutions based on pattern matching for "easy" cases (although we show that our solution is very competitive even for those). By means of a compiler-independent context-sensitive Value Set Analysis (VSA) tailored specifically to complicated indirect control transfers, we instead aim to help disassemblers handle complex and malicious code.

We compare our work against IDA Pro, a state-of-the-art pattern matching disassembler, and show that our analysis results are good and very robust. For instance, since IDA does not have good patterns for `clang`, our results are orders of magnitude better for `clang` and comparable for `gcc` even though we never embedded any compiler knowledge. In summary, our contributions are the following:

- We systematize how modern compilers implement switch statements by means of jump tables.
- We show that jump table detection by pattern matching is limited.
- We describe a context sensitive VSA suitable for recovering indirect jumps from binary code that outperforms powerful tools like IDA Pro and is compiler-independent.
- We evaluate our approach and show that it recovers complicated jump tables in binary code without access to source code or debug symbols.

2 The Problem with Patterns

Modern disassemblers commonly classify all sorts of code fragments by way of pattern matching—scanning the binary code for templates of known language constructs. For example, solutions like Jakstab and IDA Pro use well-known patterns for a variety of compilers to identify function entry points, function parameters, C++ virtual calls, switch statements, and many other constructs [34, 32, 30, 1]. Unfortunately, the effectiveness of pattern matching depends on the completeness and soundness of the templating for the code under analysis. For instance, Bao et al. [13] demonstrated the

ineffectiveness of pattern matching for detecting function entry points. In general, pattern matching does not work well if the code deviates from the templates—a common phenomenon in hand-written assembly or malware.

In this section, we systematize how modern compilers implement switch statements by means of jump tables. We then show the limitations of pattern matching for identifying these jump tables.

2.1 Jump tables in practice

Instead of a straightforward if-then-else implementation, modern compilers frequently opt for jump tables to implement switch statements [44]. In practice, compilers generate three different types of jump table instances in terms of the control flow. These types are orthogonal to Cifuentes and Van Emmerik's expressions [20] and cover all jump tables that implement switch statements in compiler-generated code that we encountered, across hundreds of applications, a wide range of compilers, and various architectures.

```
1  // compare r3 with 10
2  cmp     r3, #10
3  // if less or same, load pc
4  // with pc + value in jump
5  // table, using r3<<2 as index
6  ldrls   pc, [pc, r3, lsl #2]
7  b       default
8  .word 0x20
9  .word 0x40
10 .word 0x80
11 .word 0x40
12 ...
```

Listing 1.1: `jumpSIMPL`: `gcc` implementation of switching. An alternative implementation replaces line 1 with `subs r4, r3, #10` which changes the pattern so that IDA cannot detect it.

```
1  add     r1, r1, #1
2  and     r3, r1, #0xff
3  cmp     r3, #0xB  // 12 cases
4  mov     r1, #6
5  strb    r3, [r4,#4]
6  addls   pc, pc, r3, lsl #2
7  b       loc_7d0c  //default case
8  b       loc_7d0c  //default case
9  b       loc_7c9c  //case 1
10 b       loc_7ccc  //case 2-9
11 ...
12 ---
```

Listing 1.2: `jump2JUMP` case. Line 7 computes he value of the target. Unlike `jumpSIMPL`, it uses unconditional relative jumps instead of jump tables (lines 7-11).

`jumpSIMPL` is the most common form of jump table. It uses a register as an index in the table and computes the value of that register using the switch input value. It then loads the value of an offset from the jump table, adjusts it and adds it to the program counter. An example of this idiom is shown in Listing 1.1.

`jump2JUMP` represents an implementation that is slightly less common, but still widely used. It first adds an offset based on the switched value to the current Program Counter (PC). The new PC will target another jump (forward) instruction. The offsets are not stored in code, but in the branch forward instructions. Even though it uses no jump table in the strict sense of the word, we still consider this case for our experiments, since the computation of PC represents a significant and similar hurdle for static disassemblers. Listing 1.2 shows an example of the `jump2JUMP` idiom.

`jump2STUB` , a less common implementation, makes the code to jump to a stub that takes as parameters the switched value and the jump table. The jump table is stored af-

---

23% of switch statements are lowered to jump tables by `gcc`. When compiling SPEC CPU 2006 with `clang` (for ARM), 21% of the switch statements are lowered to jump tables.

```
1   ldrb    r3, [r4, #7]                  // r3 is the index
2   adds    r0, #0x49
3   bl      rt_switch_stub                // switch 7 cases
4
5   .byte 6                               // item count
6   .byte 0x4, 0x8, 0xd, 0x12, 0x17, 0x20
7   .byte 0x1d                            // default offset
8
9   rt_switch_stub:                       // jt width = 8 bits
10  ldrb    r12, [lr, #-1]                // load the item count
11  cmp     r3, r12                       // compare the index
12  ldrccb  r3, [lr, r3]                  // load case offset
13  ldrcsb  r3, [lr, r12]                 // load default offset
14  add     r12, lr, r3, lsl#1            // add the offset
15  bx      r12                           // jump to target
```

Listing 1.3: `jump2STUB` case. The stub uses the link register (`lr`) to access the jump table. The jump table contains the number of cases as the first entry. The default case is the last item in the jump table.

ter the unconditional jump instruction. Listing 1.3 shows an example. While less common, we did encounter this switch statement implementation on multiple occasions in ARM Thumb code, in position independent code, and in firmware. The advantage of `jump2STUB` is its space efficiency—the `rt_switch_stub` is present only once in the binary regardless of the number of switch statements.

JTR is generic enough to recover all three cases even though we do not embed any logic that models these three types. As we shall see, we do use them for evaluation.

2.2 Pattern matching limitations

Disassemblers try very hard to detect switch statements (so they know which bytes to disassemble), by matching the bytes in binary code to well-known patterns that compilers are known to generate. Any deviation from the known patterns confuses the detection. Unfortunately, it is hard to find patterns that allow jump table detection to be both sound and complete. As a result, disassemblers can easily get it wrong. Consider Listing 1.1, which shows one of the idioms generated by `gcc` to implement switch statements. A mere replacement of the `cmp` compare instruction with any semantically similar instruction such as `sub` breaks the pattern recognition even though the program semantics remain unchanged. State-of-the-art disassemblers such as IDA miss the modified jump table entirely and interpret all the data in lines 6–10 as instructions instead.

As shown in Table 1, it is quite easy to fool modern disassemblers and decompilers by deviating from such well-known patterns, but the question is whether such cases also occur in real-world code. Unfortunately, they do. For instance, the last column of Table 1 contains code that is generated by `clang`. Moreover, Listing 1.4 displays a real-world (hand optimized) implementation of the `memcpy` function in `glibc`. Note that link-time optimisation (LTO) may easily inline such highly optimized code in several places in a program. As explained in the figure, state of the art disassemblers cannot compute the target address at line 5 and line 10, because they expect the calculation of jump targets immediately before the jump itself.

	`// gcc`
	`// default`
	`cmp r3, #11`
	`ldrls pc,`
	`  [pc,r3,lsl #2]`
	`b __default`
		`// cmp->subs`
		`subs r0, r3, #11`
		`ldrls pc,`
		`  [pc,r3,lsl #2]`
		`b __default`
			`cmp r3, #11`
			`// pc alias`
			`addls r3, r3, #1`
			`ldrls r0,`
			`  [pc,r3,lsl #2]`
			`movls pc, r0`
			`b __default`
				`cmp r3, #11`
				`// redundant`
				`// cond. jump`
				`bhi __default`
				`ldrls pc,`
				`  [pc,r3,lsl #2]`
				`b __default`
					`//clang`
					`//default`
					`add r0, r0, #9`
					`cmp r0, #6`
					`bhi __default`
					`lsl r0, r0, #2`
					`add r1, pc, #0`
					`ldr pc, [r0, r1]`

IDA sw	✓	✗	✗	✗	✗
IDA CFG	✓	✓	✗	✓	✗
JTR	✓	✓	✓	✓	✓

Table 1: *JTR* Pattern matching failures. In all cases except the baseline, IDA fails to detect the switch statement ("IDA sw"). Often, this leads to an incomplete CFG also ("IDA CFG"). *JTR* always recovers the correct targets of the switch statement. The last and first column of the table is code generated by compilers.

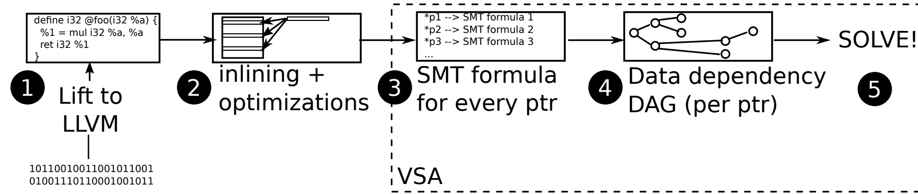

Fig. 1: High-level overview of the approach

As a result, the analysis generates an incomplete CFG which renders subsequent analysis techniques less effective—hurting, for instance, the strength of security measures that rely on binary rewriting. Likewise, reverse engineering the code now requires significant manual annotation and analysis. In the remainder of this paper, we show that *JTR* can complement pattern matching approaches and solve these cases.

3 Tailored Value Set Analysis for Solving Indirect Jumps

As shown in Figure 1, our analysis starts by lifting the binary to LLVM intermediate code using a home-grown translator, much like PIE [21] and LLBT[46], but slightly more advanced. As we do not consider it a contribution of this paper, will not discuss it further. Next, in Step 2, we apply a variety of optimizations, in particular aggressive inlining. As we will discuss later, without it LLVM does not inline some of the more intricate examples of `jump2STUB`. We now describe the main analysis steps of *JTR*—Steps 3–5 in Figure 1. Analogous to how bounded address tracking [43, 32, 33] targets

https://github.com/cojocar/bin2llvm

```
1   2:  subs    r2, r2, #96
2       [...]
3   5:  ands    ip, r2, #28  // set flags
4       rsb     ip, ip, #32  // no flag change
5       addne   pc, pc, ip   // flags are tested
6       b       7f
7   6:  nop
8       ldr     r3, [r1], #4 // r3=*r1; r1 += 4
9       [...]                // load 6 more regs
10      add     pc, pc, ip   // ip from line 5
11      nop
12      nop
13      str     r3, [r0], #4 // [r0]=r3; r0 += 4
14      [...]                // store 6 more regs
15      bcs     2b           // jump to loop entry
```

Listing 1.4: Code snippet of the implementation of `memcpy` in glibc. Note that (a) both conditional and unconditional instructions compute the targets (lines 5 and 10), (b) the condition of the `add` on line 5 is determined by the `and` instruction on line 3), and (c) the target computed on line 10 depends on a value computed 11 instructions earlier. Since most disassemblers assume locality (the calculation of jump targets right before the jump), they fail to recover this case. In contrast, *JTR* successfully computes the possible values written to the PC on lines 5 and 10.

VSA [10] at binaries, we compute a list of all possible values a register may contain at specific points in *any* program—with an emphasis on indirect control transfers.

To analyse all the indirect control transfers of interest (i.e., jump tables and complex arithmetic computations on the PC), we need only consider a program's non-constant writes to the program counter. In Step 3 in Figure 1, after identifying all such indirect writes (stores) to PC, *JTR* goes through every function containing them to determine all possible paths from the store instruction back to the start of the function. For each path, we build a set $\mathscr{C}$ that represents the specific path constraints in SMT expressions form. Specifically, we go back along the paths to discover where this value originated and stop when we encounter a memory read or the start of the function. If the memory access itself depends on an indirect memory access, we recursively trace that back also, ensuring that we handle cases where, say, the program computes a pointer p by adding pointer q and index i.

To do so, *JTR* computes a *data dependency Directed Acyclic Graph (DAG)* to capture the relation between the memory pointers:

1. A node in the graph corresponds to a memory pointer access in its SMT expression form. Because the SMT formula stops when we encounter a memory read, the expression kept in non-root nodes always contains a memory read. The SMT expression captures any complex expression between nodes.
2. An edge in the graph captures the dependency between nodes. Given two nodes p and q, an edge from p to q means that p depends on the value pointed by q. In other words, to solve pointer p, we must compute the value pointed by q. In this way, the expression of p can easily emulate (but is not limited to) an indirect memory load with a base (node q) and an offset which can be a constant or another node.
3. The root of the dependency graph represents the pointer used in the targeted indirect write and the root expression will give us the possible values of PC.

For the final step, solving the DAG, the naive approach is to invoke the Satisfiability Modulo Theories (SMT) solver for the expression of the leaves, constrained by $\mathscr{C}$. Using the obtained values, we can then subsequently load the values pointed to and solve the rest of the tree. Doing so always gives results that are an overapproximation of the real jump targets, but with a high false positive rate in case of translation imprecision.

The key observation for improving the naive solution is that the possible values of a pointer are a limited subset of all possible memory addresses and $\mathscr{C}$ and some expressions of the nodes from the DAG *must* have a common expression. Let $\mathscr{N}$ be the set of the expressions of all nodes in the DAG. We denote $\mathscr{M}$ the set of common non-constant expressions between $\mathscr{C}$ and $\mathscr{N}$. We now construct $\mathscr{M} \leftarrow [m_0, ..., m_k]$ as a sorted set, with m_0 the *largest* expression in the set. We define the size of an expression as the number of nodes needed to represent the expression as a tree.

We now ask the Z3 [40] SMT solver for concrete values for m_0 while obeying the path constraints (see Algorithm 1). Using the concrete values, we recursively solve the DAG by temporarily expanding $\mathscr{C}$ with constraints that capture the concrete values. In the second part of Algorithm 1, we start from the leaves of the DAG and we simplify each node expression using the accumulated constraints. If a node's expression becomes constant, we load its corresponding memory pointer, otherwise we continue with the simplified expression. If the memory pointer is invalid, we abandon m_i and we move to m_{i+1} and restart the process. If the expression of the root node becomes constant then we successfully solved the DAG for one value. We continue the process until all the values of m_0 are tested.

Algorithm 1 DAG solving

Require: $\mathscr{C}, \mathscr{M}$
1: **procedure** SOLVE_DAG
2:   **for** $m \in \mathscr{M}$ **do**   ▷ $\mathscr{M}$ is an ordered set
3:     **for** value∈ SMTSolve(m, $\mathscr{C}$) **do**
4:       constraint←m≡value
5:       $\mathscr{C} \leftarrow \mathscr{C} \cup$constraint
6:       rootExpr←RecursiveDAGSolve(DAG.root, $\mathscr{C}$)
7:       **if** isConstant(rootExpr) **then**
8:         appendSolution(*rootExpr*)
9:       $\mathscr{C} \leftarrow \mathscr{C} \setminus$constraint

1: **procedure** RECURSIVEDAGSOLVE(Node, $\mathscr{C}$)
2:   expressions = {}
3:   **for** child←Node.children **do**
4:     childExpr←RecursiveDAGSolve(child, $\mathscr{C}$)
5:     expressions←expressions∪(child, childExpr)
6:   **for** child, childExpr←expressions **do**
7:     **if** isConstantAndLoadable(childExpr) **then**
8:       value←LoadPointer(childExpr)
9:       constraint←childExpr≡value
10:       $\mathscr{C} \leftarrow \mathscr{C} \cup$constraint
11:   **return** simplifyExpression(Node.expr, $\mathscr{C}$)

If we explored all paths but found no solution, our analysis fails. In Section 4, we will see that despite its simplicity this method is quite effective in solving jump tables (and other indirect jumps).

Recovered code preparation As LLVM optimizations may influence our results, we evaluated the effect of important optimizations that we applied to the lifted LLVM code. As a baseline, we used the same level of optimization as in PIE [21] which already provides common optimizations such as memory to register promotion, global value numbering, and dead code elimination. Next, we added a custom pass to replace the

intricate control flow of the `select` instruction with a simpler if-then-else sequence. Finally, we turned on aggressive inlining.

In practice, presumably because the `select` instructions does not affect the control flow of the code of interest, we could not observe any change in the solving capabilities of *JTR*. Becuase *JTR* analysis is intra-procedural, aggressive inlining, *improved* our results overall as subtler `jump2STUB` were inlined and, in consequence, analyzed. We therefore turn on aggressive inlining in Step (2) of Figure 1 and in all experiments in Section 4.

4 Evaluation

We evaluate our solution on 109 coreutils programs compiled for ARM, 4 firmwares, 17 malware samples, a synthetic set of 210 binaries, and the SPEC CPU 2006 test suite. We believe that this is a meaningful set to evaluate *JTR*, as it is large-scale, contains binaries generated with different (known and unkwon) compilers, while SPEC is commonly used by the security community for benchmarking. We summarize the results in Table 2 and discuss them in detail below.

Coreutils binaries It is a clear that if accurate patterns *are* available, we cannot beat pattern matching, but we show that we are competitive still with the most important state-of-the-art dissambler. As mentioned, we intend *JTR* to *complement* rather than *compete* with traditional jump target detectors—to resolve the complicated cases that pattern matching cannot handle. Nevertheless, it is interesting to evaluate our solution by itself. To show the limitations of pattern matching and the genericity of *JTR*, we use two different compilers, namely `Clang` (version 3.5) and `GCC` (version 4.9.2). We use the debug symbols in combination with IDA to generate a "ground truth".

In the absence of debug information, IDA recovers 77% of all the switch statements. The missing 23% are either due to failed function detection, or misinterpretation of jump tables as instructions (as is the case for each of our synthetic test programs). In contrast, *JTR* recovers 98%. However, we will compare *JTR* solely with our ground truth, so as to measure against the best of what IDA could do (when IDA has the debug symbols). We believe that comparing IDA's results on stripped binaries, even though the results look better, is less meaningful. We run our analysis on an Intel(R) i7-3770 CPU based machine with 20GB of RAM on which *JTR* took 7 seconds on average per input binary and 755 seconds in total.

The results in Table 2 show that regardless of the compiler in use, *JTR* yields good results. *JTR* outperforms IDA when the `Clang` compiler is used. This is mainly because IDA uses a pattern that is usually generated by `gcc`. Specifically, the code commonly generated by `clang` for a jumptable is `ldr pc, [rX, rY]`, which is different from Listing 1.1. Moreover, `rX` and `rY` can be any general purpose register and the index value can reside in either. Coming up with a pattern that matches `Clang`'s behavior and has a low false positive rates is difficult, demonstrating the benefits of *JTR*'s generic technique.

Results on SPEC CPU test suite We again compiled SPEC with both `clang` and `gcc`. The missing cases from the SPEC benchmark are either due to compilation er-

	Coreutils		SPEC		Firmware	Malware	Synthetic
Compiler	gcc	clang	gcc	clang	unknown	unknown	various
Input binaries	109	109	12	16	4	17	210
Ground Truth	–	–	–	828	–	–	80
IDA + symbols	642	0	655	11	66	205	80
JTR	629 (97.98%)	295	573 (87.48%)	763 (92.14%)	65 (98%)	166 (81%)	80 (100%)

Table 2: *JTR* results for different test sets. The ratios in the last rows relate to the ground truth when available, and to the "IDA + symbols" row otherwise.

rors (`perlbench`, `omnetpp` and `dealII` with `clang`), or to translation errors. We instrument the `clang` compiler to generate the ground truth. However, due to code inlining after the instrumentation, this ground truth is an underapproximation of the number of switch statements actually generated. For the testcases compiled with `gcc`, we rely on IDA's output for the ground truth (given the debug symbols).

We observe the same behavior as in the case of coreutils: *JTR* succeeds both on `clang` and on `gcc` and pattern matching yields poor results on SPEC with `clang`.

To show the impact of our analysis on the quality of the CFG, for the `clang` test set, we incorporate the recovered switch statements in IDA. Due to the 9097 new edges in the CFG discovered by *JTR*, we add a cumulated 2523 basic blocks to the CFGs. The detailed results are given in Appendix 1.

Firmware Next, we evaluate *JTR* on the firmware of four different devices: a smart meter, a boot-ROM used by LPC214, a GPS stick and a GSM modem. The firmwares were manually reverse engineered in IDA, no symbols were available for this test. The ground truth is represented by the manual reverse engineering process. Our translator covered 66 switch statements that were implemented with jump tables, of which *JTR* identified all but one in the unoptimised LLVM bitcode. The missing jump table is recovered when aggressive inlining is enabled. We found 4 `jump2STUB` switch statement implementations in this set.

Malware In this experiment, we used 17 malware binaries from 7 different families: AESddos, GoARM, PnScan, Taidra, Tsunami, Elknot and LightTaidra. We manually unpacked each of the samples and then fed them to *JTR*. In practice, none of the malware samples seemed to use control flow obfuscation.

Out of the 205 switch statements identified by IDA in the translated functions, the translator *JTR* recovered 166 (81%). The main cause for this modest result is the translator: several indirect jumps are wrongly translated or completely missed, therefore the input LLVM code for *JTR* is inaccurate. Interestingly, while investigating the results on this set, we also found the bug listed in Listing 1.5. The bug was confirmed by the developers of the uClibc library.

Synthetic binaries In our next experiment, we again demonstrate that our solution is compiler agnostic by running *JTR* on 210 binaries generated from 10 C source code

"bugfix: ARM: memset.S: use unsigned comparisons"–http://goo.gl/5NiXJq

```
memset:
  mov     r3, r0
  cmp     r2, #8
  blt     2f:   // branch if < signed
  ...
2:
  movs    r2, r2
  moveq   pc, lr
  rsb     r2, r2, #7
  add     pc, pc, r2,lsl#2
// IDA dissasembler stops here
  nop
  strb    r1, [r3],#1 // repeated 8 times
  ...
```

Listing 1.5: A real-world bug found by *JTR*. This code is handcoded assembly and part of the `memset` function in uClibc. It treats the length parameter (`r2`) as a signed value. If `r2` is interpreted as a negative number, the value written to the `PC` is outside of the mapped memory.

files that contain switch statements or control flow based on jump table. We generate the binaries using 20 different compilers and compiler optimization levels from 6 different toolchains and IDEs.

In addition, we evaluated 10 cases of *hand-coded assembly*, which are not reported in Table 2 all of which were successfully recovered. *JTR* successfully recovers all of the 70 switch statements generated by the various compilers and reproduced by the translator .

4.1 Detailed analysis results

Jump table types distribution We show the distribution of the different types of jump table, as identified by IDA, in Table 3. The `jumpSIMPLE` type is the one that is by far the most popular on ARM, regardless of the test set. `jump2STUB` is rare on normal binaries but much less so in the firmware test set. In the synthetic set, we generated the `jump2STUB` cases by selecting Thumb mode and Cortex-M0 as the target platform. This CPU is often used in embedded devices, therefore compiler flags play an important role in evaluation of tools alike *JTR*. Table 3 shows that the performance of *JTR* is similar, regardless of the jump table type.

Completeness and bug finding On the ARM processor architecture, the code transitions between ARM mode and Thumb mode by means of a jump to an odd address with a specific instruction. Depending on the path, the address computed at runtime can be odd or even. When *JTR* computes the possible address value, the reported value can therefore also be either odd or even, depending on the path. The two results are essentially the same (modulo the mode) and we ignore the last bit. The computation is not ARM specific, but rather arises from the generality of the solution, as *JTR* explores both paths (ARM and Thumb).

As shown in Listing 1.5, *JTR* helps to find memory access violations. However, this is not its main objective and care must be taken when applying it naively. Specifically, because our method is (a) conservative – any pointer that fails to load on a specific path

Test set	Total	`jumpSIMPLE`	`jump2JUMP`	`jump2STUB`
Coreutils-gcc IDA	642	642	0	0
Coreutils-gcc *JTR*	629	629	0	0
Coreutils-clang IDA	0	0	0	0
Coreuitls-clang *JTR*	295	N/A	N/A	N/A
SPEC-gcc IDA	655	655	0	0
SPEC-gcc *JTR*	573	573	0	0
SPEC-clang IDA	11	11	0	0
SPEC-clang *JTR*	763	> 11	N/A	N/A
Firmware IDA (translated)	66	58	4	4
Firmware *JTR*	65	58	3	4
Malware IDA	205	152	53	0
Malware *JTR*	166	120	46	0
Synthetic binaries IDA	80	59	21	20
Synthetic binaries *JTR*	77	56	21	20

Table 3: Results of *JTR* on different jump table types. IDA is used to categorize the jump tables whenever possible.

invalidates that path, and (b) intra-procedural, the false positive rate for a bug finding strategy that uses *JTR naively* will be high. However, one may augment *JTR* with model checking techniques (e.g., specify a range of values that one register can have) to reduce the false positive rate or target only a specific family of bugs, such as stack-based buffer overflows.

4.2 Comparing *JTR* with other solutions

We tried to compare *JTR* with a variety of other solutions.

Angr The Angr framework [48] supports ARM architecture and uses static analysis to solve some jump table. Its public version (48998c5) does not work with switch statements implemented with jump tables [2]. Again, Angr generates an incomplete CFG, as the jump table targets are missing.

Jakstab While JakStab [32] does not support ARM, we tried to compare *JTR* with JakStab's public version by adapting our examples to the x86 architecture. Instead of using a switch statement, we used a table of pointer to functions. With optimizations turned off, Jakstab recovers the targeted functions. When we turn on optimizations (-O2 or -O3), its analysis fails to recover the targets.

RetDec The Retargetable decompiler [24, 35, 5] which does not use any VSA techniques, fails to retrieve targets in the absence of debug symbols. The decompiler either interprets the jump table as code or it does not reference it at all.

Radare2 Radare2's [4] support for switch statement implemented with jump tables is work in progress [3]. Note, however, that the implementation is based on pattern matching and therefore will have similar issues as IDA Pro.

REV.NG Concurrent work from Di Federico et al. [23] use VSA to analyze LLVM code to recover a complete CFG. Even so, on ARM architectures, IDA's Jackard index on CFG matching consistently outperforms REV.NG's. The results on SPEC show that *JTR* improves the quality of the CFG generated by IDA. In our experience, REV.NG performed well on simple files, but none of the configurations of SPEC binaries could currently be handled by REV.NG.

5 Related work

Jump tables and switch statements Cifuentes and Van Emmerik [20] propose a solution based on lifting the binary code to Register Transfer List (RTL) expressions. Code slicing is used to extract the expression. Next the expression are substituted until any of three known patterns are reached. The summarised patterns give enough information for recovering the possible targets of the jump table. However, the recovery of jump table's targets fails when the expression does not match one of the known patterns. Holsti [31] shows how to recover switch-case tables' targets when a Read-Only Memory (ROM) table is present. They use partial evaluation (e.g. run the program snippet with concrete input) to generate possible outputs. For this the state of the registers is modeled and loops are unrolled. This solution does not take into account the content of the memory and is dependant on detecting switch statement implementation patterns.

Meng and Miller [39] observe the difficulty of recovering an accurate CFG because of jump tables. They define three models for jump tables usage and populate these models by means of static analysis. We believe that these models are a form of pattern matching and that are not effective on ARM architecture, for example the `jump2STUB` case would require information about the where in the code the stub is. Gedich and Lazdin [29] uses the linearity property of jump tables' contents to detect them. Their solution assumes that the position of the jump table is roughly known. Once few targets are discovered, *JTR* can make use of this heuristic to accelerate the full jump table discovery.

Wang et al. [53] propose a solution to find data to code references. Their solution is working only when pointers to functions are stored in the jump table, in a data section. The compiler stores offsets rather than function pointer in the jump tables used by switch statements.

CFG recovery Reinbacher and Brauer [43] introduce a method based on SMT for generic control flow graph recovery. They leverage forward and backward abstract program interpretation to recover indirect jump targets. As opposed to *JTR*, they do not take advantage of the program's memory contents but rather use pre- and post-conditions for program's registers which are further refined by the algorithm process. JakStab [32] uses code inlining, abstract interpretation and local constant propagation to solve jump targets. It does not work on ARM. JakStab uses Bounded Address Tracking [43] and tracks every memory access and register assignment. Updates in the abstract domain are explicitly propagated. JakStab makes a distinction between memory regions. *JTR*

We compiled SPEC with Clang and with GCC. We tried static and dynamic linking.

relies on expressions and it does not need this tracking. Moreover, in *JTR* the case when a pointer points to an unknown region is captured by the SMT expressions rather than being explicitly accounted for.

Brumley et. al [17] proposes a decompiler that uses BAP's[16] VSA to recover the CFG from the tested binary. Their focus is different that *JTR*. *JTR* focuses in recovering the target of indirect control flows while Phoenix focuses on recovering high level semantics (e.g. switch statments) once the CFG is known.

Value Set Analysis Balakrishnan and Reps [11] introduce a binary static analysis technique called VSA. They show both limitations and strengths of VSA when applied to binary reverse engineering. The method used by *JTR* extends their VSA by means of the SMT solving technique.

Brauer et. al [15] argues that SMT solving is effective to do VSA but they do not leverage the memory contents'. To achieve good performance they perform liveness analysis of the Intermediate Representation (IR). This is not required by *JTR* as the expression set contains at any time the optimal set. Bounded values and k-set analysis were previously used by Bardin et al. [14] to recover indirect jumps. They exploit the locality of the indirect target computation.

6 Limitations

Path explosion A large number of paths may exist from the targeted pointer to the start of the function. Building a dependency graph for each of them and subsequently solving it could lead to resource exhaustion. The deeper into the function's CFG the program uses the pointer, the higher the chances of running into this problem. We find that in practice, limiting the size of each path to 5 LLVM basic blocks yields good results. To further optimize the running time, *JTR* solves the paths in ascending order: from the shortest to the longest.

Code discovery The accuracy of the translator, although not a real contribution of this paper, directly influences the results of *JTR*. For instance, because the translator currently uses a static view of the program, it misses jump tables that the program populates at runtime. This is not a fundamental limitation and in future work, we will fix it by feeding back the *JTR* results. Note that dynamic jump tables are not common in benign software, but it is not hard to imagine that future malware will make use of it, as an additional defense.

Memory layout and program correctness *JTR* assumes that the input code is correct and that a memory map is available. While we can extract the memory map automatically using heuristics (e.g., read/write ordering [56]), guaranteeing the correctness of the program is hard. Conversely, *JTR* can be instructed to find bugs. For instance, in Listing 1.5 we show one example in which *JTR* finds a previousely unreported bug (see also Section 4). We are confident that we can extend *JTR* to find good candidates for memory violation errors.

During the analysis, we should take special care when loading pointers that point to Input/Output (IO) memory. We cannot predict the value returned by load from an IO

memory. The naive solution is to ignore the memory accesses to IO memory and treat them as invalid accesses. However, doing so may have a negative impact on the true positive rate of *JTR* in case an IO value is used to index a jump table.

Memory aliasing Finally, the memory accesses generated by the LLVM translator can alias. When this happens the accuracy of the expressions stored in the nodes of the graph and of the path constraints decreases. The underlying reason is that *JTR* does not capture the aliasing information in SMT expressions. As future work, we will leverage the alias analysis already provided by LLVM to detect these cases.

LLVM Translator Like SecondWrite [7] and PIE [21], *JTR* builds on top of a binary-to-LLVM translator. The translator lifts the code to LLVM in a straightforward manner and *JTR* then analyzes the resulting code together with the memory image of the binary. Specifically, it uses weak heuristics for determining whether a function is in ARM mode or Thumb mode and occasionally misclassifies them. In addition, the translator does not itself resolve the indirect jump targets and its recursive descent disassembly therefore misses code fragments. The solution for the latter problem would be to feed the results of the *JTR* analysis back to the translator to discover the targeted code, but doing so is a major engineering task, and we leave this for future work.

Misclassifying a fragment's mode (ARM or Thumb) and missing code fragments in the recursive descent both cause *JTR* to miss indirect jumps and hence the appropriate targets. We stress that these issues are a problem of the translator only and not of the *JTR* analysis. By construction, *JTR* will generate a solution for the targets of every indirect jump in its input.

7 Conclusion

Jump tables on RISC architectures lead to frequent interleavings of (jump table) data and code in binaries. Most disassembler use pattern matching to detect such jump tables in binary code, which easily fails for complicated indirect control transfers. We argue that in specific security-relevant domains (handwritten code, firmware and malware), we need a more generic technique to handle the cases that elude common pattern matching. This paper proposed such a technique for "solving" jump targets for indirect control transfers. By transforming the targets to formulas that we solve in an SMT solver, we remove dependencies on templates, compilers, and processor architectures. The results show that our technique approaches and sometimes improves that of popular disassemblers that use pattern matching. *JTR* is available as an open source project: `https://github.com/cojocar/jtr`.

8 Acknowledgments

We thank the anonymous reviewers for their feedback. This work was supported by the Netherlands Organisation for Scientific Research through the grant NWO 628.001.006 CYBSEC "OpenSesame" and through the grant NWO 639.023.309 VICI "Dowsing".

References

1. IDA F.L.I.R.T. Technology: Overview.
2. Angr, Switch Statement Analysis 106. https://github.com/angr/angr/issues/106, June 2016.
3. Radare2, Analyze jump tables 3201. https://github.com/radare/radare2/issues/3201, June 2016.
4. Radare2, Portable reversing framework. https://radare.org, June 2016.
5. Retargetable Decompiler. https://retdec.com/decompilation-run/, June 2016.
6. ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. Control-flow integrity. In *CCS12* (2005).
7. ANAND, K., SMITHSON, M., ELWAZEER, K., KOTHA, A., GRUEN, J., GILES, N., AND BARUA, R. A compiler-level intermediate representation based binary analysis and rewriting system. In *ECCS8* (2013), pp. 295–308.
8. ANAND, K., SMITHSON, M., KOTHA, A., ELWAZEER, K., AND BARUA, R. Decompilation to compiler high IR in a binary rewriter. Tech. rep., University of Maryland, 2010.
9. ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN* (2014).
10. BALAKRISHNAN, G., AND REPS, T. Analyzing memory accesses in x86 executables. In *Compiler Construction* (2004), Springer, pp. 5–23.
11. BALAKRISHNAN, G., AND REPS, T. WYSINWYX: What You See is Not What You eXecute. *ACM Trans. Program. Lang. Syst. 32*, 6 (Aug. 2010), 23:1–23:84.
12. BANSAL, S., AND AIKEN, A. Binary Translation Using Peephole Superoptimizers. OSDI'08.
13. BAO, T., BURKET, J., WOO, M., TURNER, R., AND BRUMLEY, D. Byteweight: Learning to recognize functions in binary code. In *USENIX Security 14*.
14. BARDIN, S., HERRMANN, P., AND VÉDRINE, F. Refinement-based CFG reconstruction from unstructured programs. VMCAI'11, Springer.
15. BRAUER, J., HANSEN, R. R., KOWALEWSKI, S., LARSEN, K. G., AND OLESEN, M. C. Adaptable Value-Set Analysis for Low-Level Code. In *SSV'12*.
16. BRUMLEY, D., JAGER, I., AVGERINOS, T., AND SCHWARTZ, E. J. BAP: A Binary Analysis Platform. CAV'11.
17. BRUMLEY, D., LEE, J., SCHWARTZ, E. J., AND WOO, M. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. USENIX SEC'13.
18. CASTRO, M., COSTA, M., MARTIN, J.-P., PEINADO, M., AKRITIDIS, P., DONNELLY, A., BARHAM, P., AND BLACK, R. Fast byte-granularity software fault isolation. SIGOPS'09.
19. CHA, S. K., WOO, M., AND BRUMLEY, D. Program-Adaptive Mutational Fuzzing. S&P'15.
20. CIFUENTES, C., AND VAN EMMERIK, M. Recovery of jump table case statements from binary code. In *Program Comprehension* (1999).
21. COJOCAR, L., ZADDACH, J., VERDULT, R., BOS, H., FRANCILLON, A., AND BALZAROTTI, D. PIE: Parser Identification in Embedded Systems. ACSAC 2015.
22. DAVI, L., LEHMANN, D., SADEGHI, A.-R., AND MONROSE, F. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. USENIX SEC'14.
23. DI FEDERICO, A., PAYER, M., AND AGOSTA, G. Rev.Ng: A Unified Binary Analysis Framework to Recover CFGs and Function Boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, ACM, pp. 131–141.

24. DURFINA, L., KŘOUSTEK, J., ZEMEK, P., KOLÁ\VR, D., HRUSKA, T., MASARÍK, K., AND MEDUNA, A. Design of a Retargetable Decompiler for a Static Platform-Independent Malware Analysis. *International Journal of Security and Its Applications 5*, 4 (2011), 91–106.
25. ERLINGSSON, U., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. C. XFI: Software guards for system address spaces. OSDI'06.
26. EVANS, I., LONG, F., OTGONBAATAR, U., SHROBE, H., RINARD, M., OKHRAVI, H., AND SIDIROGLOU-DOUSKOS, S. Control jujutsu: On the weaknesses of fine-grained control flow integrity. CCS'15.
27. FORD, B., AND COX, R. Vx32: Lightweight User-level Sandboxing on the x86. In *USENIX Annual Technical Conference*.
28. GANESH, V., LEEK, T., AND RINARD, M. Taint-based Directed Whitebox Fuzzing. ICSE '09.
29. GEDICH, A., AND LAZDIN, A. Improved Algorithm for Identification of Switch Tables in Executable Code. FRUCT'15.
30. HARRIS, L. C., AND MILLER, B. P. Practical analysis of stripped binary code. *ACM SIGARCH Computer Architecture News 33*, 5 (2005), 63–68.
31. HOLSTI, N. Analysing Switch-Case Tables by Partial Evaluation. In *WCET* (2007).
32. KINDER, J., AND VEITH, H. Jakstab: A Static Analysis Platform for Binaries. CAV '08.
33. KINDER, J., AND VEITH, H. Precise Static Analysis of Untrusted Driver Binaries. FMCAD '10.
34. KÄSTNER, D., AND WILHELM, S. Generic Control Flow Reconstruction from Assembly Code.
35. KŘOUSTEK, J. *Retargetable Analysis of Machine Code*. PhD thesis, Faculty of Information Technology, Brno University of Technology, CZ, 2015.
36. LI, Y., MCCUNE, J., NEWSOME, J., PERRIG, A., BAKER, B., AND DREWRY, W. Minibox: A two-way sandbox for x86 native code. *USENIX ATC 14*.
37. MCCABE, T. J. A complexity measure. *Software Engineering, IEEE* (1976).
38. MCCAMANT, S., AND MORRISETT, G. Evaluating SFI for a CISC Architecture. USENIX-SS'06.
39. MENG, X., AND MILLER, B. Binary code is not easy. ISSTA'16.
40. MICROSOFT. The Z3 Theorem Prover. https://github.com/Z3Prover/z3, February 2016.
41. MING, J., WU, D., XIAO, G., WANG, J., AND LIU, P. TaintPipe: Pipelined Symbolic Taint Analysis. In *USENIX SEC'15*.
42. O'SULLIVAN, P., ANAND, K., AND KOTHA, A. Retrofitting security in COTS software with binary rewriting. IFP SEC'11.
43. REINBACHER, T., AND BRAUER, J. Precise control flow reconstruction using boolean logic. In *EMSOFT 2011*.
44. SAYLE, R. A. A superoptimizer analysis of multiway branch code generation. In *Proceedings of the GCC Developers Summit* (2008).
45. SEHR, D., MUTH, R., BIFFLE, C. L., KHIMENKO, V., PASKO, E., YEE, B., SCHIMPF, K., AND CHEN, B. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *USENIX SEC'10*.
46. SHEN, B.-Y., CHEN, J.-Y., HSU, W.-C., AND YANG, W. LLBT: An LLVM-based Static Binary Translator. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems* (New York, NY, USA, 2012), CASES '12, ACM, pp. 51–60.
47. SHIN, Y., AND WILLIAMS, L. An Empirical Model to Predict Security Vulnerabilities Using Code Complexity Metrics. ESEM '08.

48. SHOSHITAISHVILI, Y., WANG, R., SALLS, C., STEPHENS, N., POLINO, M., DUTCHER, A., GROSEN, J., FENG, S., HAUSER, C., KRUEGEL, C., AND VIGNA, G. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *S&P'16*.
49. SMITHSON, M., ANAND, K., AND KOTHA, A. Binary rewriting without relocation information. Tech. Rep. November, University of Maryland, 2010.
50. TICE, C., ROEDER, T., COLLINGBOURNE, P., CHECKOWAY, S., ERLINGSSON, U., LOZANO, L., AND PIKE, G. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *USENIX SEC'14*.
51. TIKIR, M. M., LAURENZANO, M., CARRINGTON, L., AND SNAVELY, A. PMaC Binary Instrumentation Library for PowerPC/AIX. In *Workshop on Bin. Inst. and App.* (2006).
52. VAN DER VEEN, V., ANDRIESSE, D., GÖKTAŞ, E., GRAS, B., SAMBUC, L., SLOWINSKA, A., BOS, H., AND GIUFFRIDA, C. Practical context-sensitive cfi. CCS'15.
53. WANG, S., WANG, P., AND WU, D. Reassembleable Disassembling. In *USENIX SEC'15*.
54. WANG, X., JHI, Y.-C., ZHU, S., AND LIU, P. Still: Exploit code detection via static taint and initialization analyses. In *ACSAC'08*.
55. YEE, B., SEHR, D., DARDYK, G., CHEN, J., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *S&P'09*.
56. ZADDACH, J., BRUNO, L., FRANCILLON, A., AND BALZAROTTI, D. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. NDSS'14.
57. ZENG, B., TAN, G., AND MORRISETT, G. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *CCS'18* (2011), ACM, pp. 29–40.
58. ZHANG, M., AND SEKAR, R. Control Flow Integrity for COTS Binaries. In *USENIX SEC'13*.

1 SPEC result details

Testcase	Clang	IDA	*JTR*	Edges added	BBs added	IDA	*JTR*
h264ref	26	0	25 (96.15%)	142 (0.794%)	10 (0.079%)	17	17 (100.00%)
soplex	27	0	40 (148.15%)	241 (1.641%)	20 (0.183%)	36	40 (111.11%)
cactusADM	36	0	34 (94.44%)	1399 (6.819%)	766 (5.116%)	34	29 (85.29%)
gromacs	40	0	40 (100.00%)	367 (1.685%)	30 (0.192%)	43	41 (95.35%)

Table 4: IDA results on SPEC binaries compiled with `clang` are depicted in the first 5 columns. The first column represents the number of switch statement as reported by `clang`. We instrumented `clang` to tell if a switch statement was lowered to a jump table before code inlining takes place, thus the above 100% success rate on some cases for *JTR*. IDA misses most of the jump tables on `clang`. Column 3 and 4 show how the CFG benefits from the newly discovered targets. The percentages are relative to the total number of edges and basic blocks. The results for SPEC when compiled with `gcc` are shown in the last two columns. Here *JTR* performs better than IDA on `soplex`.