Speculative disassembly of binary code

M. Ammar Ben Khadra
University of Kaiserslautern
khadra@eit.uni-kl.de

Dominik Stoffel
University of Kaiserslautern
stoffel@eit.uni-kl.de

Wolfgang Kunz
University of Kaiserslautern
kunz@eit.uni-kl.de

ABSTRACT

Embedded software is rapidly increasing in complexity. To cope with this, developers rely on third-party IPs to accelerate product delivery. However, IP source code might not be available which limits verifiability. This creates a particular challenge especially in safety-critical applications, e.g., automotive. Static Binary Analysis (SBA) is a promising technique to address such a challenge by providing engineers with the ability to reason about the actual instructions executed for all possible inputs. Disassembly is the fundamental first step for any SBA where assembly instructions are recovered from binary code. Correct disassembly, however, is challenging since data is mixed with code in binaries. Moreover, variable-size ISA, e.g., Thumb and TriCore, allow a single byte sequence to have multiple valid interpretations.

We introduce Spedi, an open source SPEculative DIsassembler for Thumb ISA. Spedi is based on a principled approach to disassembly where all possible basic blocks are speculatively recovered. Then, basic blocks are refined using conflict analyses to identify assembly instructions. Experiments using a wide range of benchmarks demonstrate that Spedi is both fast and effective. It outperforms IDA Pro, the de-facto industry standard disassembler, in terms of disassembly correctness. Spedi can also recover the majority of the call graph and switch table targets. It is resilient to obfuscation and doesn't use any symbol information which makes it a suitable front-end for a wide variety of SBA applications including security analysis.

CCS Concepts

•Social and professional topics → Software reverse engineering; •Software and its engineering → *Automated static analysis; Translator writing systems and compiler generators;* Formal software verification;

Keywords

disassembly; reverse engineering; binary analysis

CASES '16, October 01-07, 2016, Pittsburgh, PA, USA
© 2016 ACM. ISBN 978-1-4503-4482-1/16/10...$15.00
DOI: http://dx.doi.org/10.1145/2968455.2968505

1. INTRODUCTION

Complexity of embedded software is rapidly increasing. This makes embedded software providers depend on third-party IPs, e.g., libraries, to fill the "innovation gap" and to accelerate product delivery. However, third-parties might not be willing to share the source code of their IPs to keep their competitive advantage. This can create verification issues during software integration that are hard to trace. Without source code, engineers are basically left with testing and dynamic trace analysis which can miss bugs. In safety-critical applications, e.g., automotive, it is highly desirable to accelerate the pace of innovation while maintaining software correctness at the same time.

Static Binary Analysis (SBA) is a promising technique to address those challenges. Basically, SBA provides the ability to reason about precise instruction behavior for all possible inputs. It has many important applications including formal verification [19, 18], static instrumentation [1, 24], equivalence checking [7, 21], and security analysis [2, 6]. The latter is particularly important as more nodes become connected to the "cloud" which, consequently, exposes more targets, e.g., cars [4], to potential cyber attacks. Moreover, SBA can be essential even when source code is available. Some analyses such as static timing analysis and security analysis can be effectively carried out only at binary level. Finally, compilers might introduce bugs [12, 23] or unexpected side effects [2] that can only be detected by reasoning about the actually executed instructions rather than the source code.

Disassembly is the fundamental first step for any SBA where assembly instructions are recovered from binary code. Correct disassembly, however, is challenging since data bytes are mixed with instructions code in binaries. Additionally, variable-size ISA, e.g., Thumb[1] and TriCore, allow for multiple valid interpretations for the same byte sequence. It is possible to use symbol information, e.g., relocation symbols, to assist in correct instruction recovery. However, symbols may not be available. Even when available, they might be not trustworthy or even incorrect [8]. Therefore, disassemblers in safety-critical applications, e.g., [20], depend on pattern matching of known compiler idioms which requires knowing the compiler, or maybe even the exact compiler version, used to produce the binary. Basically, our contributions can be summarized as follows:

- A principled method for disassembly which is compiler independent and doesn't utilize symbol information.

[1]We use the term Thumb to refer to Thumb-2 ISA, which is the variable size extension to Thumb-1 introduced in ARMv7.

- CFG recovery techniques that are (mostly) ISA independent. Our techniques focus on indirect branches found in procedure returns and switch tables[2].
- An open source tool, Spedi, which efficiently implements our approach for ARM's Thumb ISA.

1.1 The disassembly problem

The disassembly problem, a.k.a. code discovery problem, can be summarized as follows: given a buffer of bytes C_{buf} characterized by (a_{st}, s, a_{en}) where a_{st} is the start address, s is the size of C_{buf}, and a_{en} is the address of the entry execution instruction such that $a_{st} \leq a_{en} < a_{st} + s$ holds. The goal is to recover the following without using any extra information:

- Instruction recovery. Recovery of the set I_v of valid instructions. An instruction is valid if it is possibly reachable when starting execution from a_{en}.
- CFG recovery. Recovery of directed graph $G = (V, E)$ such that for basic blocks $u, v \in V$ there is an edge $e \in E : u \to v$ iff v is reachable from u.

The challenge is that code is typically mixed with data in the instruction stream which potentially results in data bytes being incorrectly disassembled as instructions. Additionally, instructions in variable-size ISA, e.g., Thumb and TriCore, can be 2 or 4 bytes in size. Hence, incorrect disassembly of data can result in valid instructions being missed which can also affect static analysis results. The problem is even more challenging in CISC ISA, e.g., x86 where instruction size can vary between 1 and 15 bytes.

The traditional algorithms of disassembly are *linear-sweep* and *recursive-descent*. In the former, disassembly starts at a_{st} and then follows the byte stream till the end of C_{buf}. That is, for each disassembled instruction t_i disassembly of t_{i+1} starts at address $start(t_{i+1})$ such that,

$$start(t_{i+1}) = start(t_i) + size(t_i)$$

Linear sweep is fast and simple, however, it often yields a significant number of disassembly errors. Consider for example the byte sequence that starts at 0xb992 in Fig.1 where each assembly instruction is shown next to its start address. Linear sweep incorrectly diassembles the instructions at 0xb998 and 0xb99a despite being data. Upon reaching 0xb99c it disassembles rsb.w which is of 4 bytes size. Consequently, disassembly continues at 0xb9a0 skipping the valid instruction at 0xb99e.

Data bytes do not affect processor execution since they are not reachable at run time. Recursive descent disassembly tries to mimic that behavior by starting from $a_{en} =$ 0xb992 and following the CFG. However, upon reaching the indirect Control Transfer Instruction (CTI) at 0xb996, the algorithm would stop following that path since it can't know the possible values that r2 can take at run time. However, it still follows the conditional CTI at 0xb992 and correctly continues disassembly at the target address 0xb99e. Recursive descent disassembly is correct as long as it encounters direct CTIs. However, a significant part of the binary can be reachable only through indirect CTIs which can lead to low code coverage. In practice, disassemblers in the industry, e.g., IDA Pro [16], mix both disassembly methods together with custom heuristics to achieve high code coverage.

[2]We refer here to the recovery of the target addresses of jump tables produced by compiling switch statements in languages like C.

b992: 1b b1	cbz	r3, b99e	cbz	r3, b99e
b994: 01 99	ldr	r1, [sp, #4]	ldr	r1, [sp, #4]
b996: 10 47	bx	r2	bx	r2
b998: 48 b0	add	sp, #0x120	*.data*	
b99a: 02 00	movs	r2, r0	*.data*	
b99c: cb eb	rsb.w	fp, fp, r1	*.data*	
b99e: 01 0b			lsrs	r1, r0, #0xc
b9a0: 39 46	mov	r1, r7	mov	r1, r7
b9a2: 04 22	movs	r2, #4	movs	r2, #4
b9a4: af e7	b	b906	b	b906

Figure 1: linear sweep vs. correct disassembly

Our proposed disassembly method provides complete code coverage together with correct instruction recovery. The work flow of our method is depicted in Fig. 2. We take an ELF binary as input and recover the set of all potential basic blocks. A basic block (BB) is a sequence of instructions that ends with a CTI. A CFG is then built based on direct CTIs. *Instruction recovery* is based on the refinement of that CFG using overlap and CFG conflict analyses. Later, we attempt to recover targets of indirect CTI in the *CFG recovery* phase based on call-graph and switch-table recovery. Our method is based on the following insights:

- Data bytes can only exist after CTIs and before the beginning of valid BBs. That is, they can't exist within the instruction sequence of a valid BB.
- Valid BBs are more connected to the CFG compared to spurious BBs generated by speculative disassembly. This gives them more "weight" that we use to resolve conflicts with invalid BBs.

Our discussion continues as follows: in Sec. 2 we discuss the steps needed to recover assembly instructions from a binary. In Sec. 3 we consider the recovery of indirect CTI targets. The focus is on the targets of switch tables and procedure returns. We assume that the binary is generated by a well-behaving compiler and/or assembler. However, this assumption might be invalid if the analyzed binary was obfuscated by a third party. Resilience to obfuscation techniques is discussed in Sec. 4. Then, we discuss our experimental results in Sec. 5. Finally, we refer to related work in Sec. 6.

2. INSTRUCTION RECOVERY

Instruction recovery is separated into three steps which are speculative disassembly, overlap conflict analysis, and CFG conflict analysis.

2.1 Speculative disassembly

Our goal in this step is to determine all potential BBs available in a binary, i.e., to recover a superset of valid BBs. To this end, we start disassembly at a_{st} and continue till the end of C_{buf} which is similar to a linear sweep. However, we speculatively attempt to disassemble every possible instruction. In Thumb this would be at every 2 bytes. In comparison, this would be at every byte in x86. Unlike recursive-descent disassembly we do not use a_{en}. Every disassembled instruction is appended to one or several corresponding BBs if they already exist, otherwise, a new BB is created for that instruction. BBs that meet at the same CTI are grouped in the same Maximal Block (MB). In Fig. 1, BBs starting at 0xb99c and 0xb99e are grouped in the MB that ends with the CTI at 0xb9a4. An MB is a convenient container for all BBs that exhibit the same control-flow behavior.

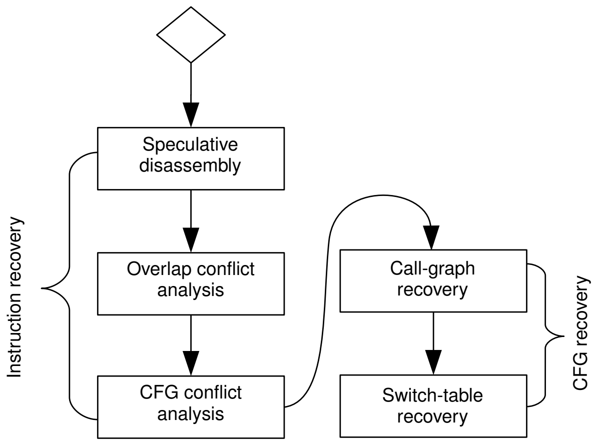

Figure 2: Work-flow of proposed method

We introduce some necessary notation, for an instruction t its end address is $end(t) = start(t) + size(t)$. Note that $start(t)$ is unique for each t. Also, for a basic block bb, it holds that $end(bb) = start(bb) + size(bb)$ where $start(bb) = start(t_0)$ such that t_0 is the first instruction in bb. $size(bb)$ is the sum of sizes of its instructions, and $end(bb) = end(t_i)$ such that t_i is the last instruction in bb. We say that t can append bb iff $end(bb) = start(t)$. For a maximal block mb the sets T_{mb} and B_{mb} represent the sets of instructions and basic blocks it contains, respectively. Also, $end(mb) = end(t_{cti})$ where t_{cti} is its CTI. We say that a maximal block mb_{i+1} can append mb_i iff:

$$\exists bb \in B(mb_{i+1}) \mid end(mb_i) = start(bb)$$

Maximal blocks are recovered using Algorithm 1. Basically, we decode every possible instruction t (line #4). Then, we check if t is acceptable. An instruction is acceptable if (1) its bytes are decodable, and (2) it doesn't violate ISA rules. For example, there are certain restrictions on the usage of register pc in ARM ISA manuals that, if violated, can make such instruction unexecutable. Then, we check if t represents a CTI (#line 6). If not, we look into the current MB for possible BBs that t can append (line #11). If none is found a new BB is created containing t only.

If t is a CTI then it is appended like any other instruction. However, an MB has to be constructed at this moment since at least one BB has become complete. Basically, BBs contained in the current MB are classified into two main categories (1) complete BBs which have been appended by the CTI, or (2) incomplete BBs which hold a sequence of instructions but without a CTI. Note that a complete BB contains a single CTI. Later, incomplete BBs are classified into two sets, (1) invalid BBs, and (2) overlap BBs. Overlap BBs are kept since they can potentially be appended by an instruction in the next MB. For a BB to be regarded as an overlap BB it should satisfy the following:

$$end(mb) - end(bb) < S_{max}$$

where S_{max} is the maximum size of an instruction. In the case of Thumb ISA it is 4 bytes. All overlap BBs are kept for the next MB (#line 9) while other BBs are discarded since a valid BB can't exist without a CTI. Algorithm 1 has linear complexity except for BB classification (line #9). The cost of classifying BBs as complete, invalid, or overlap

Algorithm 1: Maximal block recovery

```
Input   : Bytes to be disassembled $C_{buf}$
Output: List of maximal blocks $\Gamma_{mb}$
1  $addr := a_{st}$;
2  $mb :=$ create_maximal_block();
3  while $addr < end\_of(C_{buf})$ do
4      inst := decode_inst_at($addr$);
5      if $acceptable(inst)$ then
6          if $is\_cti(inst)$ then
7              append($mb$, $inst$);
8              add_to_result($\Gamma_{mb}$, $mb$) ;
9              $mb :=$ get_overlap($mb$);
10         else
11             append($mb$, inst);
12         end
13     end
14     $addr := addr + 2$;
15 end
```

Algorithm 2: Build direct CFG

```
Input   : List of maximal blocks $\Gamma_{mb}$
Output: Direct CFG $\Gamma_{cfg}$
1  foreach $mb$ in $\Gamma_{mb}$ do
2      $inst :=$ get_cti($mb$);
3      if $is\_direct\_cti(inst)$ then
4          $mb_r =$ find_remote_target($inst$);
5          add_edge($\Gamma_{cfg}$, mb, $mb_r$);
6          if $is\_conditional\_cti(inst)$ then
7              $mb_i =$ find_immediate_target($inst$);
8              add_edge($\Gamma_{cfg}$, mb, $mb_i$);
9          end
10     end
11 end
```

is linear $O(|B_{mb}|)$. However, classifying instructions that belong to either BB class is $O(|B_{mb}| * |T_{mb}|)$, in our current implementation.

THEOREM 1. *The set of basic blocks recovered in Algorithm 1 is a superset of valid basic blocks existing in C_{buf}.*

Proof. Algorithm 1 disassembles every possible instruction (line #14). Let bb be a valid basic block. Then, bb can either consist of a single t_{cti}. In such case it's recovered directly in a maximal block. Otherwise, bb consists of a sequence of instructions T_{bb} that ends with a t_{cti} such that $\forall t_i, t_{i+1} \in T_{bb}, end(t_i) = start(t_{i+1})$. Algorithm 1 recovers all instruction sequences by checking for each disassembled t whether it appends a current sequence, otherwise, a new T_{bb} will be constructed containing t only (line #11). Upon disassembling a t_{cti} then either (1) it appends T_{bb} which means that bb is completed in the current maximal block (line #8), or (2) it is a spurious t_{cti} which means that T_{bb} will be retained for the next maximal block (line #9). $\square$

Having recovered the list of MBs Γ_{mb} we build the CFG Γ_{cfg} using only direct CTI as depicted in Algorithm 2. Basically, it's a single pass through MBs where an edge is added to Γ_{cfg} between mb and mb_r if the latter is reachable using a direct CTI (line #5). Another edge should be added for conditional CTI (line #8) iff mb_i can append mb. Finding

Algorithm 3: Overlap conflict resolution

```
Input  : CFG $\Gamma_{cfg}$
Output: Modified CFG $\Gamma_{cfg}$
1  foreach $mb$ in $\Gamma_{cfg}$ do
2      if has_overlap(mb) then
3          $mb_o$ = get_overlap_node( $mb$);
4          if is_shrinkable($mb_o$) then
5              shrink_if_applicable($mb_o$);
6          else
7              invalidate_either($mb$, $mb_o$);
8          end
9      end
10 end
```

a remote target (lines #4) returns a result only if the target MB actually contains an instruction with the designated start address, otherwise, the procedure fails. This is similar for the procedure of finding an immediate target (line #7). Failing to find a target means that the MB, and all of its direct predecessors, should be discarded since Algorithm 1 provides a guarantee of recovering a superset of valid BBs. However, when analyzing dynamically linked executables we allow an MB to target an external ELF section, e.g., .plt section, but we make sure that the targeted section is executable. The control flow can't jump to a section holding data, e.g., .rodata section.

Algorithm 2 is linear in $|\Gamma_{mb}|$. Finding an immediate target (line #7) is typically $O(1)$ since it's the immediate successor in $|\Gamma_{mb}|$. The most costly operation is the binary search required to find a remote MB (line#4). In this search we take advantage of the ordering found in $|\Gamma_{mb}|$ which can be stated as:

THEOREM 2. *Let mb_i and mb_j be maximal blocks in $|\Gamma_{mb}|$ such that $i < j$ then $end(mb_i) \leq end(mb_j)$.*

Proof. Consider how MBs are recovered in Algorithm 1. Specifically, consider how new MBs are added to $|\Gamma_{mb}|$ (line #8). Each MB ends with a single CTI where $end(mb) = end(t_{cti})$. Given that no two CTIs can start at the same address, then, we have $start(t^i_{cti}) < start(t^j_{cti})$ since instructions are disassembled in ascending order starting from a_{st}. This leads to $end(t^i_{cti}) \leq end(t^j_{cti})$ since t^j_{cti} may overlap with t^i_{cti} in the last 2 bytes. □

2.2 Conflict analysis

We recovered in the previous step a superset of valid BBs and their direct CFG. We attempt here to refine our BBs such that for each MB there should be a single valid BB. Note that a BB can be uniquely identified by its start address. Therefore, we attempt to choose the "best" start address for each MB.

2.2.1 Overlap conflict analysis

Consider Algorithm 1 where the current maximal block, mb, in line #9, retains overlap BBs. An overlap BB can be appended with a CTI in the next MB to form a complete BB. In that case, we say that both MBs overlap. Formally, for mb_i and mb_j to overlap where $i < j$ the following holds,

$$start(mb_j) < end(mb_i)$$

This situation creates a conflict that should be resolved since valid instructions can not overlap. To resolve such

```
2cb08:  c2 f1 | rsb.w0  r2, r2, 1f |
2cb0a:  1f 02 |                    |
2cb0c:  20 fa | lsr.w0  r0, r0, r2 |
2cb0e:  02 f0 |                    | and1   r7, r2, f0000000
2cb10:  70 47 | bx0     lr         |
2cb12:  08 b1 |                    | cbz1   r0, 2cb18
2cb14:  4f f0 | mov.w2  r0, #-1    |
2cb16:  ff 30 |                    |
2cb18:  00 f0 | b.w2    2cf0c      |
2cb1a:  f8 b9 |                    | cbnz3  r0, 2cb5c
2cb1c:  00 29 | cmp4    r0, #0     |
2cb1e:  f8 d0 | beq4    2cb12      |
```

Figure 3: Overlap conflict example

conflict we can either (1) invalidate mb_i, or (2) shrink mb_j. Shrinking an MB means increasing its start address such that $end(mb_i) \leq start(mb_j)$. In resolving an overlap we take into account the following heuristics:

- Alignment. If mb_j can be appended to some mb_k, with $k < i$, then it should be easier to invalidate mb_i than to shrink mb_j.
- Connectedness. If mb_i is "more connected" to the CFG than mb_j then it should be easier to shrink mb_j than to invalidate mb_i.

Both heuristics, alignment and connectedness, are based on a parameter called "weight". In our implementation, the weight of an MB is the sum of its own instruction count and the instruction counts of its first-level predecessors.

Algorithm 3 implements our overlap analysis. Basically, an overlap mb_o is checked if *shrinkable* (line #4), if not, we have to choose between mb and mb_o based on their weight. Note that mb_l is shrinkable iff,

$$\exists t \in T^l_{mb} \mid end(mb_l) \leq start(t)$$

On the other hand, if mb_l is shrinkable then we set its start address to the first t that satisfies the above relation. Shrinking mb_l effectively invalidates some instructions at its beginning which is not always applicable. For example, another mb_k in Γ_{cfg} could be targeting one of those invalidated instructions which leads to a conflict between mb and mb_k that should be resolved using their weight.

We discuss overlap analysis using an example from du[3] depicted in Fig. 3. It depicts 4 MBs where each instruction is annotated with the index of its MB. We have two overlap conflicts in total. The first is between mb_0 and mb_1 where $end(mb_0) =$ 0x2cb12 $<$ $start(mb_1) =$ 0x2cb0e. The second conflict is between mb_2 and mb_3 where $end(mb_2) =$ 0x2cbc2 $<$ $start(mb_3) =$ 0x2cb1a. Resolving the first conflict is done by shrinking mb_1 such that the instruction at 0x2cb0e is invalidated. As for the second conflict, it's not possible to shrink mb_3 since it consists of a single instruction. Consequently, mb_3 is invalidated.

2.2.2 CFG conflict analysis

The goal of overlap conflict analysis is to adjust the start address of each overlapping MB to an instruction that doesn't share any bytes with a previous MB. Having resolved such *inter-MB* conflicts we discuss here resolving *intra-MB* conflicts between different BBs. Basically, each BB starts at a

[3]One of the tools found in GNU's Coreutils.

9dec: 10 b5	push_0	{r4, lr}			
9dee: 41 f2	mov.w_0	r4, 1268			
9df0: 68 24				movs_1	r4, 68
9df2: c0 f2	$\text{movt}_{0,1}$	r4,#3			
9df4: 03 04				lsls_2	r3, [r4]
9df6: 23 78	$\text{ldrb}_{0,1,2}$	r3, [r4]			
9df8: 1b b9	$\text{cbnz}_{0,1,2}$	r3, 9e02			

Figure 4: CFG conflict example

unique start address within its MB. Therefore, only one BB might be a valid BB while others are spurious. Consider the MB depicted in Fig. 4 which ends with a conditional CTI at 0x9df8. It consists of 3 BBs starting at addresses 0x9dec, 0x9df0, and 0x9df4, respectively. Each instruction is annotated with the BB(s) it belongs to. For example, bb_0 and bb_1 meet at instruction 0x9df2 and henceforth share all the following instructions.

Let's assume that overlap analysis has set the start address of this MB to 0x9dec. However, as far as overlap is concerned any address greater than 0x9dec is acceptable. For example, choosing to start at 0x9df0 means that instructions at 0x9dec, 0x9dee, and 0x9df4 have to be invalidated since they do not belong to the valid BB. Invalidating an instruction would also mean invalidating any MBs that could be targeting it in Γ_{cfg}. Such a conflict is resolved by weighing each BB and choosing the BB with highest weight. The weight of a BB is calculated similarly as in the case of MB weight. That is, the sum of its instruction count together with the instruction count of its direct predecessors.

3. CFG RECOVERY

We attempt in this section to complete Γ_{cfg} by recovering the targets of indirect CTIs. Precise recovery of indirect CTI targets is generally undecidable [9]. However, binaries produced by compilers are *structured*, a fact that we try to exploit here. Generally, indirect CTIs in binary code can be classified into the following:

- Procedure returns. A branch from callee to caller.
- Switch tables. Intra-procedural branch using an index.
- Indirect calls. A call to a procedure that is resolved at run time.

Next, we discuss the recovery of call graph and switch tables. We do not consider indirect calls since they require a more complicated points-to analysis.

3.1 Call graph recovery

The call graph is a directed graph where each node represents a procedure[4] and each edge is a call between two procedures. Recovering the call graph enables identifying the targets of procedure returns since a callee should, typically, go back to its caller. However, the call graph is also important for any context-sensitive analysis.

We recover the call graph by partitioning Γ_{cfg} to a set of sub-graphs $\Gamma^i_{cfg} = (e^i_{mb}, M^i_{mb}, A^i, E^i_{mb})$ where e^i_{mb} is the entry MB, M^i_{mb} is the set of MBs that belong to the procedure, A^i is the area that M^i_{mb} occupies in memory. In our procedure model, we assume A^i to be contiguous. $E^i_{mb} \subseteq M^i_{mb}$ are *exit* MBs where their branch target is outside A^i.

[4]We do not differentiate between the terms procedure, function, and subroutine in this context.

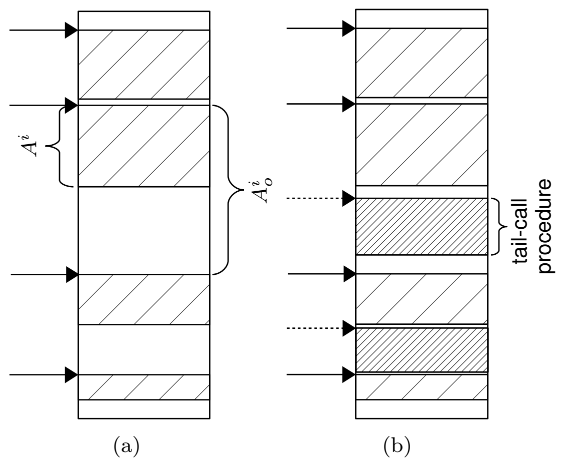

Figure 5: Call graph recovery (a) directly called procedures, (b) after tail-calls identification.

For static analysis purposes we are interested in a *well-formed* procedures which are more amenable to *summarization*. For Γ^i_{cfg} to be well-formed it should satisfy (1) all $mb^i \in M^i_m$ should be dominated by e^i_{mb}, i.e., only reachable through e^i_{mb}, (2) all exits $mb^i \in E^i_{mb}$ should either be a *call* to another well-formed procedure or a *return* to caller. Unlike Fortran, C and C++ do not allow multiple entries for a procedure which means that condition #1 generally holds except for subroutines written in assembly. Additionally, note that a procedure call can either be *return-call* (henceforth a call) where the callee returns to its direct caller by saving its return address, or a *tail-call* where the callee doesn't necessarily return to its immediate caller. For example, it might return to another caller up in the stack frame. ARM specifies instructions bl and blx to be used for calls where the return address is saved to the link register lr before branching. Tail-calls are implemented using usual branch instructions, e.g., b and bx. Similar calling conventions apply to other ISAs. Moreover, note that procedures in source code do not necessarily map directly to their binary counterparts due to procedure inlining especially in link-time optimizations.

Our method for procedure identification proceeds in two phases. Firstly, call instructions in Γ_{cfg} are used to identify procedure entries and memory area. That is done by performing a pass on Γ_{cfg} where the targets of MBs that branch using a direct call are added to set L_e. That is, if mb^i is branching using a call then $target(mb^i) \in L_e$. Then, procedure entries in L_e are sorted and used to segment the address space of C_{buf}. By now, we have, for each directly called Γ^i_{cfg}, identified its e^i_{mb} and an overapproximation A^i_o of its A^i as depicted in Fig. 5a. Now we traverse each procedure using Depth-First Search (DFS) starting from its e^i_{mb} where each unvisited mb^i is added to M^i_{mb}. Additionally, we track reads and writes of lr during procedure traversal. That enables correct identification of return exit points. For example, a procedure that pushes lr to specific index on the stack should use that value upon returning.

Additionally, we check the branch target of each mb^i to be within A^i_o before continuing traversal. In case of branching outside A^i_o then mb^i will be added as an exit point to E^i_{mb}. Traversal stops at exit points and at indirect CTIs except for switch statements which we discuss in the next section.

```
int switch_test(int var){
switch(var){
case 2:
var+=3; break;
case 5:
var+=2; break;
case 9:
var+=5; break;
case 11:
var+=4; break;
default:
var+=10;
}
return var;
}
```

(a) source code

```
0x0c:  cmp     r3, #11
0x0e:  bhi.n   60
0x10:  add     r2, pc, #4
0x12:  ldr.w   pc, [r2, r3, lsl #2]
0x18:  .data   0x00000040
0x1c:  .data   0x00000060
0x20:  .data   0x00000060
0x24:  .data   0x00000048
0x28:  .data   0x00000060
0x2c:  .data   0x00000060
0x30:  .data   0x00000060
0x34:  .data   0x00000050
0x38:  .data   0x00000060
0x3c:  .data   0x00000058
0x40:  ldr     r3, [r7, #4]
0x42:  adds    r3, #3
```

(b) absolute address table

```
0x2:   cmp     r3, #11
0x4:   bhi.n   24
0x6:   tbb     [pc, r3]
0xa:   .data   0x0d07
0xc:   .data   0x0d0d050d
0x10:  .data   0x090d0b0d
0x14:  movs    r0, #7
0x16:  bx      lr
0x18:  movs    r0, #5
0x1a:  bx      lr
0x1c:  movs    r0, #15
0x1e:  bx      lr
0x20:  movs    r0, #14
0x22:  bx      lr
0x24:  adds    r0, #10
0x26:  bx      lr
```

(c) address offset table

Figure 6: An sample code with 2 different switch table implementations

The exit point with the maximum $end(mb^i)$ is used to mark the actual traversed area A^i. Note that a direct CTI outside A^i_o can either target the entry MB of a tail-called procedure, which forms a *tail-call* exit, or an MB in the body of another procedure, which is an *overlap* exit. The distinction between overlap and tail-call exit points enable modeling procedures with multiple entries. Also, it simplifies other analyses.

The second phase of our identification method attempts to identify tail-called procedures in addition to unreachable procedures. The latter are only called indirectly or maybe not called at all. We perform a pass through Γ_{cfg} and for each mb that was not assigned to a procedure in the previous phase we assume that it is a procedure entry to be traversed. Traversal here is done similarly as in the previous phase. However, the approximated memory area A^i_o for a procedure is now limited to areas that are not held by procedures previously identified as depicted in Fig. 5b.

A final issue to consider are calls to non-returning functions, e.g, abort. The problem here is that we typically assume that a call is returning. Therefore, we continue traversal starting from the immediate successor. That might lead us to erroneously step into another procedure. We address this by identifying calls to common non-returning procedures based on a map between procedure addresses in .plt section and their corresponding names recovered from dynamic symbols. After that, we identify internal non-return functions by analyzing their exit points.

3.2 Switch table recovery

In recovering the call graph, the MBs of each procedure were identified by CFG traversal starting from its entry MB. Our traversal is based on the fact that intra-procedural CTIs are mostly direct. However, switch tables represent the exception to this rule since they are usually implemented using indirect CTI. That is, we do not know the branch targets of a switch table although we know that they are typically intra-procedural. The challenge in switch tables is that their implementation in binaries can vary depending on the compiler (version), optimization level, and number of case statements it has. For example, a switch statement that has three cases might more efficiently be implemented using a series of conditional CTIs. Our focus here is on switch statements that are implemented using indirect CTIs.

Consider Fig. 6 where two different switch table implementations were generated for the same code using the same compiler but with different optimization settings. In Fig. 6b, the switch table was implemented using absolute addresses. Basically, *index* value r3 is first checked if higher than *limit* in order to branch to the default case. If not higher, then r2 should hold the value of the switch table's *base* which is 0x18. Then, the control flow branches indirectly by loading the target address into pc using a load instruction at 0x12. Each word in the switch table starting at the base holds an absolute address of an MB. For example, the first case statement can be found at address 0x40.

In comparison, a different implementation was generated in Fig. 6c. The base of that table is 0xa where each byte is used to calculate an address offset in tbb . For example, byte at address 0x10 = 0x9 is used as follows,

$$\begin{aligned} target &= base + 2 * offset \\ &= 0xa + 2 * 0x9 \\ &= 0x1c \end{aligned}$$

While actual switch-table implementations vary, they generally adhere, semantically, to a "canonical" form [5],

```
if(index > limit)
  pc = default_case
else
  pc = jump_func(base, index)
```

Based on that, an architecture-independent method to identify the targets of switch tables should be capable of:

- Identifying the value of the switch table's base.
- Identifying possible run time values of the index.
- Compute the jump function.

Cifuentes et al. [5] implemented such a method to identify possible switch branch targets. They implement backward program slicing starting from the switch table's CTI. This process stops at the procedure's entry or when the value of the index is influenced by external input. Then, the sliced instructions are analyzed. Note that their analysis is implemented after translating instructions to an Intermediate Representation (IR). Otherwise, such an analysis would be cumbersome to implement. To be effective, they also use ISA-specific knowledge of switch table types.

Our method is more lightweight in comparison and doesn't involve IR analysis, yet it is still effective in practice. It can be summarized in Algorithm 4. Basically, we consider every indirect CTI as a candidate switch table. That can be

Algorithm 4: Switch table target recovery

```
Input  : $\Gamma^p_{cfg}$ MBs inside procedure's memory area
Output: Modified $\Gamma^p_{cfg}$
1 foreach mb in $\Gamma^p_{cfg}$ do
2     inst := get_cti(mb);
3     if is_indirect(inst) then
4         ⟨type, base⟩ = try_backward_slicing(inst);
5         if is_valid(type) then
6             compute_switch_table_targets(type, base);
7         end
8     end
9 end
```

confirmed if backward slicing was successful and able to compute the table's base (#line 4). To be successful, it has to be confirmed that the CTI depends on at least two registers for its computation and that one of those registers (base value) can be statically computed. That is, its computation depends on a static values, e.g, pc and immediate values. For example, in Fig. 6b, upon reaching the indirect CTI at 0x12 we can directly check that its computation depends on r2 and r3. Slicing backwards we find add at 0x10 where r2 can be statically computed. Note that the pc value in Thumb is $start(t) + 4$ where t is the instruction that uses pc.

Having identified the switch table's base we start computing the table targets sequentially starting from the base address (line #6). Targets are computed based on the type of switch table. We currently support the three switch table types that we found in practice, namely, (1) absolute address tables, (2) tbb byte offset tables, and (3) tbh halfword offset tables. Computing targets stops upon reaching the first case statement or hitting an invalid target. The latter happens when incorrectly decoding padding bytes. We perform additional clean-up in case an invalid target was found. Finally, note that Algorithm 4 works on Γ^p_{cfg}. That is, switch recovery happens after identifying a procedure's memory area and before actually traversing a procedure.

4. RESILIENCE TO OBFUSCATION

Obfuscation is any transformation applied to a program to make it harder to reverse-engineer. Obfuscation can be applied at source code (or similar) level to make its algorithms difficult to *decompile.* Additionally, it can also be applied to the binary code to make it difficult to *disassemble.* Our focus is on the latter techniques that are applied statically, i.e., without run-time code modification. Static obfuscation techniques try to violate assumptions that normally hold for well-behaving binaries, like (1) either branch of a conditional CTI can be taken, and (2) control flow returns to the instruction that immediately follows a call.

Moser et al. [14] explored violating assumption #1 by introducing *opaque predicates* that are provably difficult to statically analyze. An opaque predicate is an indirect conditional CTI that *always* branches to only one of its targets. In that way, junk bytes can be inserted at the address of the other target which confuses static disassemblers. Linn et al. [13] implemented an effective obfuscator that violates #2 by replacing direct calls with *branch functions* that indirectly call the required procedure at run-time.

Kruegel et al. [11] used speculative disassembly to build an effective x86 disassembler tailored towards the obfuscator

```
237ae: | ite      le
237b0: | movle    r7, r0
237b2: | bgt.w    229f8
237b6: | .data
```

Figure 7: Spurious conditional CTI

of Linn et al. [13]. Our work is inspired by the former and we share the basic ideas of speculative disassembly and refinement through conflict analysis. Hence, we provide comparable resilience to obfuscation. However, we use different data structures which significantly affects how we tackle the problem. In the work of Kruegel et al. each instruction belongs to a single *fragment*, a sequence of instructions that doesn't necessarily end with a CTI. In comparison with our method, each instruction belongs to a single MB. Due to the self-repairing property of x86 disassembly [17], a fragment would normally contain 1–3 instructions. This leads to inefficiencies in conflict analysis. We identify, in the following, key differences between our methods.

Procedure identification. Due to the data-structure inefficiencies they require that procedure entries (or their approximation) should be identified first, using an external method, such that speculative disassembly is applied to each procedure individually. We do not impose such a restriction. Instead, we recover procedures and switch tables after completing instruction recovery.

Conflict analysis. They resolve conflicts in one step by assigning a weight to each fragment and selecting fragments with highest weight. A fragment's weight is the number of its direct predecessors (sometimes also successors are used). In comparison, we do conflict analysis in two steps. Firstly, we resolve conflicts among MBs in overlap analysis. Then, we resolve conflicts among BBs in the same MB in CFG conflict analysis. This provides more efficiency due to our coarse-granular data structures. It also provides flexibility in tackling subtle cases. For example, consider the MB depicted in Fig. 7 taken from du. Its CTI is conditional only because it happens to be the else case for the invalid instruction ite at 0x237ae. In our method, ite is efficiently invalidated in overlap analysis, while in the method of Kruegel et al. the whole fragment (and its predecessors) should be invalidated since it can conditionally branch into data bytes. It would be interesting to extend our method to x86 in order to better compare both methods. Note that applying our method to x86 would require a more elaborate overlap analysis which we leave as a future work.

Statistical analysis. Gaps between procedures are speculatively disassembled in a separate step. Here, they use bi-gram statistical learning in order to weight instructions. Basically, every 2 instructions that appear more frequently together in their learning corpus should have more weight that is used to calculate the total weight of a fragment. In comparison, we do speculative disassembly uniformly for the whole code in one pass and we do not use statistical analysis.

To conclude, obfuscation and de-obfuscation are an arms race. We guarantee, through speculative disassembly, that all static BBs are recovered. How BBs are later *refined* depends on how far an obfuscator can go. Eliminating most direct CTIs is, in principle, possible which would affect conflict analyses in our method as well as that of Kruegel et al. However, this would also severely affect program performance and still not be strong enough to deter dynamic analysis.

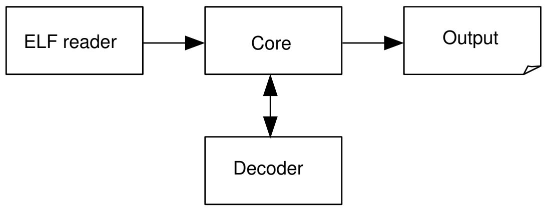

Figure 8: Tool architecture

5. EVALUATION

We begin by discussing our tool implementation and experimental setup before moving to disassembly results.

5.1 Tool implementation

Our proposed method has been implemented in Spedi[5] using C++. Spedi's architecture is depicted in Fig. 8. Basically, it consists of a reader, a binary code decoder and the core engine. The reader currently supports the widely used ELF format. However, our method is generic and is not tied to ELF or any other binary format. The decoder decodes code bytes to their corresponding assembly instructions. For this purpose, we use the Capstone library [15] , an open-source multi-architecture binary decoder. This makes our results directly applicable to other ISAs supported by Capstone, e.g., x86 and PowerPC. Our main contribution is the core speculative disassembly engine.

Our experiments have been conducted on a Linux machine with 8 virtual cores (4 physical cores) and 16 GB of RAM. We used Linux's Coreutils and CoreMark Pro benchmark suites compiled using gcc versions 4.8 and 5.1 respectively. This provides more confidence that our results are not affected by changing compilers. We selected the biggest five programs in each benchmark suite. Compilation was done with -O3 to give the compiler enough chance to "challenge" the disassembler. Our results are reported for the disassembly of .text section in each executable. Executables are dynamically linked to demonstrate that our techniques remain effective even when not all procedures are available. We used IDA Pro v6.9.1 in our experiments. Data was extracted from IDA Pro using its IDAPython API. We refer to results obtained from IDA Pro using binaries with symbol-tables by IDA^s. Otherwise, all results were obtained from stripped binaries. We restrict our discussion to key results only due to paper space limitations.

5.2 Instruction recovery

We begin by giving an overview about our benchmark binaries in Table 1. Basically, we show the number of valid instructions in addition to CTIs and indirect CTIs available. Valid instructions were identified based on ARM code mapping symbols which are available in the symbol table. We built a disassembler for that purpose and validated our ground-truth results with objdump which can also use code symbols when available. Note that ICTIs constitute a significant percentage of all CTIs which challenges pure recursive descent disassembly.

We evaluate instruction recovery based on the VIR metric which is the ratio of valid instructions recovered compared to valid instructions in the binary. Our discussion is based on Table 2. We compare VIR results by providing objdump, IDA, and Spedi with stripped binaries. Objdump uses linear sweep disassembly in the absence of symbols. Expectedly, it produces a high number of disassembly errors. Basically, it consistently achieved a VIR of about 51% which renders any SBA based on it to be effectively useless. In comparison, IDA was in a better position by achieving about 98% on average. For unclear reasons, sha was the exception with 74.53%. Inspecting error causes, we noticed that IDA was skipping valid instruction even in sequence of correctly disassembled BBs. Move instructions, e.g., movt and movs, were particularly affected by such skipping.

[5]Spedi website: https://github.com/abenkhadra/spedi.

Table 1: Benchmark statistics

	Orig.	CTI	ICTI
nnet	13749	1942	21.27%
sha	14800	1843	22.41%
parser	17059	2916	19.24%
loops	23110	2759	17.83%
zip	26789	4057	20.90%
expr	30423	6728	12.08%
ls	30460	6640	14.50%
csplit	31342	6761	11.64%
ptx	34301	7634	11.96%
du	48809	10940	12.61%

Table 2: Instruction recovery results

	objdump	IDA	Spedi	
	VIR	VIR	VIR	Time (ms)
nnet	49.37%	97.30%	99.94%	51
sha	50.13%	74.53%	99.95%	101
parser	50.00%	97.72%	99.95%	56
loops	49.98%	98.31%	99.96%	79
zip	50.08%	97.87%	99.97%	82
expr	50.47%	98.86%	99.97%	89
ls	54.00%	97.77%	99.97%	91
csplit	50.42%	98.78%	99.97%	94
ptx	50.47%	98.94%	99.97%	102
du	79.13%	98.18%	99.98%	146

On the other hand, Spedi demonstrates consistent results for all benchmarks. Actually, our main source of disassembly errors is the procedure call_weak_fn which consists of 7 ARM instructions. We do not support mixed ARM/Thumb disassembly in our implementation. Note that achieving a VIR of 100 % doesn't mean that the disassembly method is perfect since some data bytes can still be decoded as instructions. However, we already identify most data bytes which are used by switch tables and PC relative load instructions.

As for scalability, we show averaged execution time of Spedi for 10 runs. Spedi did scale gracefully with increased program size. The only exception was sha which is relatively high at 101ms. The reason behind that is sha, unusually, has some very large MBs with thousands of instructions. That makes building BBs and attaching instructions to them expensive. Finally. Spedi's time can't be directly compared to objdump or IDA due to varied functionality. Basically, objdump does a simple linear sweep while IDA needs more analysis to build a disassembly database.

5.3 Call graph recovery

Call graph recovery is a the problem of partitioning Γ_{cfg}

Table 3: Procedure recovery results

		IDA		Spedi	
	Orig.	RP	PRP	RP	PRP
nnet	296	70	1	294	2
sha	296	49	1	292	3
parser	325	102	1	321	3
loops	337	79	3	332	3
zip	377	97	2	372	4
expr	172	103	2	165	6
ls	247	114	6	233	11
csplit	174	107	4	170	3
ptx	191	114	5	186	3
du	305	175	15	291	12

to a set of sub-graphs $\Gamma^i_{cfg} | i \in \mathbb{N}$ where each Γ^i_{cfg} exhibits well-formed procedure properties. Expectedly, many procedures do not necessarily adhere to such properties. However, our procedure model is generic by allowing overlaps and tail-calls to be easily captured. Hence, such peculiarities can be coped with depending on the SBA requirements.

To measure the quality of call-graph recovery, we assess to what extent it correlates with original procedures. Specifically, to consider Γ^i_{cfg} to be a Recovered Procedure (RP), it's *entry* and *end* addresses should exactly match those of its corresponding original procedure. If only one of those addresses is matched, then we consider Γ^i_{cfg} to be a Partially Recovered Procedure (PRP). Original procedure boundaries were determined using IDA^s and double checked with the symbol table. Our results are depicted in Table 3. Clearly, Spedi significantly outperforms IDA in procedure identification both in terms of RP and PRP. That is due to the fact that our method guarantees that every MB should be assigned to a procedure.

The main sources of errors in our procedure recovery method are, firstly, one or more invalid instructions could be disassembled before the entry MB of an unreachable procedure. With the absence of a direct CTI, we can't know exactly where such procedure starts. Hence, we keep the largest possible BB. Secondly, consider the case of procedures P_i and P_{i+1}, where P_{i+1} is not called directly. If P_i have a tail-call to P_{i+1} then we will consider both of them as a single procedure which starts at P_i and ends at P_{i+1}.

5.4 Switch-table recovery

In evaluating switch-table recovery we had to cross-check switch-table results produced by IDA^s, IDA, and Spedi in order to reach a "consensus" regarding each switch-table and its switch targets. Results are depicted in Table 4. Binaries with no switch-tables were omitted. The original number of switch-tables is shown together with the number of switch-table identified by IDA^s, IDA, and Spedi, respectively.

We begin by comparing the results of IDA^s and IDA. Interestingly, IDA^s consistently identifies a superset of switch-tables identified by IDA. However, the case of du was an exception where IDA recognized two switch-tables that where missed by IDA^s. Note that the same python scripts where used in our experiments. Additionally, IDA^s successfully identified all switch-tables only in the case of parser and zip. Surprisingly, almost all switch tables where correctly identified in the GUI. Manual GUI inspection shows that only one switch-table was actually missed by IDA Pro. Consequently, IDA Pro incorrectly disassembled the data bytes following it. This switch-table used lr as a based register which is rather unusual.

Table 4: Switch-table recovery results

	Orig.	IDA^s	IDA	Spedi
parser	3	3	2	3
zip	2	2	1	2
expr	16	15	14	16
ls	20	19	17	20
csplit	16	15	13	16
ptx	18	17	15	18
du	30	28	28	30

In comparison, Spedi demonstrates consistent results in identifying all switch-tables. Actually, Spedi depends on the "semantic" of switch-tables which are inferred reliably using backward slicing. In summary, we correctly identify 105 switch-tables which have 1058 branch targets in total. Among the identified targets there was a single false-positive. Currently, we stop backward slicing upon reaching the first conditional CTI which is, typically, a comparison with the limit of index register. We need to improve our analysis based on that to better eliminate false-positives. Finally, IDA Pro is more oriented towards *interactive* disassembly driven by a human analyst. That might justify the inconsistencies we experienced between results produced by its GUI compared to IDAPython API. On the other hand, we are concerned with *automatic* SBA. That motivates even more the need to fill the gap addressed in this work.

6. RELATED WORK

The literature on binary disassembly is rich.Therefore, we will try to cover prominent works that do not depend on symbol information. Zhang et al. [24] used iterative linear sweep disassembly using objdump to implement their binary rewriter. Anand et al. [1] used speculative disassembly for code areas that are unreachable directly. Those areas would be translated to LLVM IR in order to be refined later or simply rewritten back to the binary such that they would be executed in case required at run time. Wartell et al. [22] used speculative disassembly, however, their instruction recovery depends on statistical machine learning. The closest work to ours is that of Kruegel et al. [11] which we already discussed in Sec. 4. However, their work doesn't address switch-table or call-graph recovery.

CFG recovery has been tackled from various perspectives. We start with procedure identification. A straightforward method to identify procedures begins with detecting their entries based on common patterns like stack allocations. Some proposals [3, 17] rely on machine learning to automate pattern recognition. However, procedure entries vary and not all procedures need stack allocations. Moreover, stack allocations can happen in an inner MB rather than in the entry MB in optimized code. Kinder et al. [10] approached CFG recovery from an abstract interpretation perspective. They use recursive-descent disassembly until reaching an indirect CTI. Targets of indirect CTIs are safely overapproximated using a custom abstract domain that they implemented for their Intermediate Representation (IR). In case of loops, a significant part of the CFG needs to be reanalyzed which is computationally expensive. Harris et al. [8] used recursive-descent disassembly combined with practical heuristics for function identification and CFG recovery.

Finally, Theiling [20] used recursive-descent disassembly in addition to pattern-matching of compiler idioms. Program slicing (based on an IR) was used for switch-table recovery. He identifies procedures using a top-down approach which is applicable only to statically linked programs.

Compared to other work, our approach is light-weight by not using an IR or expensive analysis. Also, it's more reliable compared to approaches that use machine learning or custom heuristics since it depends on the actual CFG. Finally, our speculative disassembly method lends itself easily to parallelization where each thread can work independently on a memory region before merging results with other threads.

7. CONCLUSIONS

We introduced Spedi, a speculative disassembler for the variable-size Thumb ISA. Spedi is based on a principled approach for speculative disassembly where a all possible basic blocks are first recovered. Then, basic blocks are refined using conflict analyses to identify valid ones. In addition, effective techniques for call graph and switch table recovery have been discussed. Our techniques are mostly compiler independent and do not rely on any symbol information. Spedi is both fast and effective as demonstrated by our experiments where it outperformed IDA Pro on stripped binaries. Consequently, it's readily applicable to industrial settings. We make Spedi, together with our benchmark binaries, publicly available for the sake of open science.

8. REFERENCES

[1] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proc. of the 8th ACM Euro. Conf. on Comp. Sys. (EuroSys '13)*, pages 295–308, 2013.

[2] G. Balakrishnan and T. Reps. WYSINWYX:What You See Is Not What You eXecute. *ACM Trans. on Prog. Lang. and Sys.*, 32(6):1–84, 2010.

[3] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *Proc. of the 23rd USENIX Security Symposium*, pages 845–860, 2014.

[4] S. Checkoway, D. Mccoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno. Comprehensive Experimental Analyses of Automotive Attack Surfaces. In *Proc. of the 20th USENIX Security conference*, pages 6–6, 2011.

[5] C. Cifuentes and M. Van Emmerik. Recovery of jump table case statements from binary code. In *Procs of the 7th Int. Workshop on Program Comprehension*, pages 192–199, 1999.

[6] M. Cova, V. Felmetsger, G. Banks, and G. Vigna. Static Detection of Vulnerabilities in x86 Executables. In *Proc. of the 22nd Annual Comp. Sec. App. Conf. (ACSAC'06)*, pages 269–278, 2006.

[7] D. Currie, X. Feng, M. Fujita, A. J. Hu, M. Kwan, and S. Rajan. Embedded Software Verification Using Symbolic Execution and Uninterpreted Functions. *Int. Journal of Parallel Programming*, 34(1):61–91, 2006.

[8] L. C. Harris and B. P. Miller. Practical analysis of stripped binary code. *ACM SIGARCH Comp. Arch. News*, 33(5):63–68, 2005.

[9] R. N. Horspool and N. Marovac. An approach to the problem of detranslation of computer programs. *Computer Journal*, 23(3):223–229, 1980.

[10] J. Kinder, F. Zuleger, and H. Veith. An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries. In *Proc. of the 10th Int. Conf. on Verification, Model Checking, and Abstract Interpretation*, volume 5403 of *LNCS*, pages 214–228. Springer, 2009.

[11] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proc. of the 13th USENIX Sec. Symp.*, pages 255–270, 2004.

[12] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *Proc. of the 35th Conf. on Prog. Lang. Design and Impl*, pages 216–226, 2014.

[13] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proc. of the 10th ACM Conf. on Comp. and Comm. Sec. (CCS '03)*, page 290, 2003.

[14] A. Moser, C. Kruegel, and E. Kirda. Limits of Static Analysis for Malware Detection. In *Proc. of the 23rd Annual Comp. Sec. Applications Conf. (ACSAC '07)*, pages 421–430, 2007.

[15] Online. Capstone disassembly library. http://www.capstone-engine.org/.

[16] Online. Hex-rays' ida pro disassembler. https://www.hex-rays.com/products/ida/.

[17] N. Rosenblum, X. Zhu, B. Miller, and K. Hunt. Learning to analyze binary computer code. In *Proc. of the 23rd National Conf. on AI (AAAI'08)*, pages 798–804, 2008.

[18] B. Schlich. Model checking of software for microcontrollers. *ACM Trans. on Embedded Comp. Sys.*, 9(4):1–27, 2010.

[19] A. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. Reps. Directed proof generation for machine code. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification*, volume 6174 of *LNCS*, pages 288–305. Springer, Berlin, Heidelberg, 2010.

[20] H. Theiling. Extracting safe and precise control flow from binaries. In *Proc. of the 7th Int. Conf. on Real-Time Comp. Sys. and Applications (RTCSA'00)*, pages 23–30, 2000.

[21] C. Villarraga, B. Schmidt, J. Bormann, C. Bartsch, D. Stoffel, and W. Kunz. An equivalence checker for hardware-dependent embedded system software. In *Proc. of 11th Int. Conf. on Formal Methods and Models for Codesign*, pages 119–128, 2013.

[22] R. Wartell, Y. Zhou, K. W. Hamlen, and M. Kantarcioglu. Shingled Graph Disassembly: Finding the Undecideable Path. In *Proc. of 18th Pacific-Asia Conf. on Knowledge Discovery and Data Mining (PAKDD'14)*, pages 273–285, 2014.

[23] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proc. of the 32nd Conf. on Prog. Lang. Design and Impl. (PLDI'11)*, pages 283–294, 2011.

[24] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Proc. of the 22nd USENIX Sec. Symp.*, pages 337–352, 2013.