An Empirical Study on ARM Disassembly Tools

Muhui Jiang
csmjiang@comp.polyu.edu.hk
The Hong Kong Polytechnic University
Hong Kong SAR

Yajin Zhou∗
yajin_zhou@zju.edu.cn
Zhejiang University
China

Xiapu Luo
csxluo@comp.polyu.edu.hk
The Hong Kong Polytechnic University
Hong Kong SAR

Ruoyu Wang
fishw@asu.edu
Arizona State University
USA

Yang Liu
yangliu@ntu.edu.sg
Nanyang Technological University
Institute of Computing Innovation,
Zhejiang University

Kui Ren
kuiren@zju.edu.cn
Zhejiang University
China

ABSTRACT
With the increasing popularity of embedded devices, ARM is becoming the dominant architecture for them. In the meanwhile, there is a pressing need to perform security assessments for these devices. Due to different types of peripherals, it is challenging to dynamically run the firmware of these devices in an emulated environment. Therefore, the static analysis is still commonly used. Existing work usually leverages off-the-shelf tools to disassemble stripped ARM binaries and (implicitly) assume that reliable disassembling binaries and function recognition are solved problems. However, whether this assumption really holds is unknown.

In this paper, we conduct the first comprehensive study on ARM disassembly tools. Specifically, we build 1,896 ARM binaries (including 248 obfuscated ones) with different compilers, compiling options, and obfuscation methods. We then evaluate them using eight state-of-the-art ARM disassembly tools (including both commercial and noncommercial ones) on their capabilities to locate instructions and function boundaries. These two are fundamental ones, which are leveraged to build other primitives. Our work reveals some observations that have not been systematically summarized and/or confirmed. For instance, we find that the existence of both ARM and Thumb instruction sets, and the reuse of the BL instruction for both function calls and branches bring serious challenges to disassembly tools. Our evaluation sheds light on the limitations of state-of-the-art disassembly tools and points out potential directions for improvement. To engage the community, we release the data set, and the related scripts at https://github.com/valour01/arm_disassembler_study.

CCS CONCEPTS
• Software and its engineering → Assembly languages.

KEYWORDS
Disassembly Tools, ARM Architecture, Empirical Study

ACM Reference Format:

1 INTRODUCTION
ARM is becoming the dominant architecture for embedded and mobile devices. At the same time, the security of these devices has attracted raising attentions [26, 34, 36, 43, 59, 61], possibly due to serious consequences if they are compromised. For instance, a botnet with hijacked IoT devices could bring down popular websites [26] and disrupt power grids [61].

The software of these devices, i.e., the firmware, is usually closed-source. Researchers have to analyze the (stripped) firmware without access to the source code or debugging symbols. Due to different types of peripherals of these devices, it is an ongoing research problem to dynamically run firmware under an emulated environment at a large scale [35, 36, 41, 67]. Because of this (sad) fact, static analysis is still a dominating methodology used by the community [34, 37, 42, 59, 65]. For instance, it has been used to locate bugs [42], find authentication bypass vulnerabilities [39] and build a general framework to rewrite ARM binaries.∗ [62].

As shown in Table 1, previous systems usually leverage off-the-shelf disassembly tools to identify instructions and function boundaries. They assume that reliably disassembling stripped binaries is a solved problem. However, whether this assumption really holds is unknown. Andriesse et al. [25] performed an analysis of disassembly tools on x86/x64 binaries and presented their findings that “some constructs, such as function boundaries, are much harder to recover accurately than is reflected in the literature.” Although the paper provides insights of the mismatch between assumptions in papers and current capabilities of popular disassembly tools, it only focuses on x86/x64 binaries. Whether their findings could be applied to ARM binaries is unknown.

∗In this paper, we use “ARM binaries” to refer to binary programs running on CPUs of the ARM architecture. An ARM binary can have both ARM and Thumb instruction sets (Section 2.1) [34].
### Challenges of disassembling ARM binaries

ARM binaries have some unique properties, which bring challenges when disassembling them. First, inline data is common in ARM binaries while "constructs like inline data and overlapping code are very rare" in x86/x64 binaries [25]. Second, ARM provides two instruction sets: the ARM instruction set and the Thumb instruction set (which includes both 16-bit Thumb-1 and 32-bit Thumb-2 instructions). An ARM binary can have both ARM and Thumb instructions. Precisely identifying the correct instruction set is challenging. Third, there is no distinguished function call instruction in ARM binaries, unlike the call instruction on x86/x64. Compilers tend to reuse other instructions, such as the branch and link instruction (BL), for both function calls and direct branches in the Thumb instruction set. This makes identifying functions more challenging. Due to these unique properties, there is a need to perform an extensive and thorough evaluation of ARM disassembly tools.

### Our work

In this work, we perform an empirical study of ARM disassembly tools. In particular, we evaluate these tools’ capabilities of two primitives, the instruction boundary and the function boundary. These two are fundamental ones that other primitives (e.g., control flow graph and call graph) are built upon. To make the study comprehensive, it should meet the following requirements:

1. Our evaluation should use diverse programs, including both popular benchmarks and, more importantly, different types of real programs that are commonly used in the wild.
2. Our evaluation should cover different compilers with various compiling options, e.g., different instruction sets and optimization levels.
3. Our evaluation should consider options of tools, which may affect the result.
4. Our evaluation should include obfuscated binaries [69], since they do exist in the wild and could affect the results of disassembly tools.

To this end, we cross-compile 1,040 real-world programs and 19 benchmark programs. In total, we get 1,896 binaries, where 608 are from the SPEC CPU2006 with different compiling options. Among the remaining ones, 1,040 are Android daemons, libraries, and user-space binaries of embedded systems (e.g., OpenWRT [19]). We also build 248 obfuscated binaries using O-LLVM [45] with multiple obfuscation methods. Then we obtain the ground truth with the help of debugging symbols and feed the stripped binaries (binaries without debugging symbols) to eight state-of-the-art ARM disassembly tools, including three commercial ones (i.e., IDA Pro [13], Hopper [12], and Binary Ninja [5]) and five noncommercial ones (i.e., Ghidra [11], arm-linux-gnueabi-objdump [18], angr [60], Radare2 [22], and BAP [31]). Finally, we measure the precision and recall by comparing the differences between the ground truth and the disassembling result.

Based on the result, we present some observations that were not systematically summarized and/or confirmed. First, the unique properties of ARM binaries do bring challenges to the disassembly tools, especially the two different instruction sets (i.e., the ARM instruction set and the Thumb instruction set) and the reuse of the BL label for both function call and branch. Second, disassembly tools do not have a good support to binaries in Thumb instruction set. The precision and recall for disassembling Thumb instructions are usually lower than that of ARM instructions (more than 90% in maximum). Third, the robustness and scalability of disassembly tools should be improved. We observed several exceptions, segment faults and timeout during the analysis. Fourth, other factors, including compilers, compiling options, target CPU architectures, could affect the result. However, the root cause is still due to the unique properties of ARM binaries.

We have reported our findings along with failed test cases to developers of the evaluated tools [14–17]. Developers of Binary Ninja, Hopper, and angr verified our findings and provided updates based on the failed cases. Radare2 assigned bug tag to them. Ghidra verified our findings and provided the potential solutions, while BAP declared that they would solve the problem in the future. To engage the community, we release the data set, and the related scripts at https://github.com/valour01/arm_disassembler_study.

In summary, this work makes the following contributions:

- We summarize the unique properties of ARM binaries that bring challenges to the disassembling of ARM binaries.
- We perform the first comprehensive study of state-of-the-art disassembly tools on ARM binaries and report our findings, which show that, contrary to the previous assumption, reliably disassembling ARM binaries is not yet a solved problem.
- Our evaluation and further analysis of failed cases reveal their root causes and provide insights and future directions for improvements.

## 2 BACKGROUND

### 2.1 CPU Architectures and Instruction Sets

ARM has multiple CPU architectures, each with different instruction extensions and features. When building binaries, developers can specify the target CPU architecture, e.g., ARMv5 or ARMv7, through compiling options (-march). For instance, ARMv5 is the default CPU architecture of the GCC compiler.

Moreover, there are two instruction sets, i.e., the ARM instruction set and the Thumb instruction set. The former is 32-bit long, while the latter is 16-bit long and designed for size-sensitive applications, which is available for ARMv4T CPU architecture and later versions. Since ARMv6T2, Thumb-2 is introduced. It offers "best of both worlds" compromise between the ARM instruction set and the Thumb instruction set. It has access to both 16-bit and 32-bit instructions. In this paper, we use the Thumb instruction set to denote both Thumb and Thumb-2 instruction encoding.

A single binary can contain multiple instruction sets and switch between them, e.g., switching between ARM instructions and Thumb (Thumb-2) instructions. The switching can occur explicitly by executing branch instructions or implicitly specified by branch targets.
Figure 1: The source code and its corresponding ARM, Thumb and Thumb-2 instructions.

For instance, the BLX $1abe1$ instruction always changes the instruction set from ARM to Thumb or vice versa. However, the BX $Rm$ derives the target instruction set from bit[0] of the register $Rm$. If it is 0, then the target instruction set is ARM. Otherwise, it is Thumb. The target instruction set of other branch instructions, e.g., POP (PC, $Rm$ . . . ), also depends on the last bit of the target address. This brings serious challenges for disassembly tools to statically determine the target instruction set, especially for the ones that leverage linear sweep strategy (Section 2.2).

Figure 1 illustrates the source code of a function and the binary compiled using ARM, Thumb and Thumb-2 instructions. Note that, for the two popular compilers (e.g., GCC and Clang), the default instruction set is ARM, and the option -mthumb is used to change it to Thumb. The instruction set greatly affects the accuracy of disassembly tool, which we will discuss in Section 4.

2.2 Disassembly Strategies

To understand the capability of disassembly tools, we use two primitives, i.e., the instruction boundary and the function boundary in our study. To precisely detect the instruction boundary, a disassembly tool should be able to locate the inline data inside the binary and the correct instruction set (ARM vs Thumb).

There are two different disassembly strategies [54, 57]. One is linear sweep, which linearly decodes the code sections. It is used by disassemblers such as the GNU utility objdump. However, the inline data (data inside the code section), and instruction set switching cannot be detected by this strategy since it does not consider the control flow transfers. Figure 2 shows a function with three basic blocks and inline data between the basic block 2 and the basic block 3. The objdump tool fails to determine the boundary between code and data, and disassembles the inline data as code.

Another strategy is recursive traversal. Its basic idea is disassembling code from the entry point of a binary, and then recording the branch targets as new entry points (usually appends these branch targets into a list). It repeats this process until no new targets could be found, and all the targets in the list have been traversed. The advantage of this strategy is that it is unlikely to disassemble inline data as code, since there should be a control flow transfer instruction before the inline data (otherwise, the data will be executed as code at runtime). Moreover, it can handle instruction set switch if the branch target can be determined statically (direct branches). However, the disadvantage is that some code regions may be missed, if they cannot be reached through direct branches. As in Figure 2, the code in the basic block 3 may be missed since this block can only be reached through an indirect branch, whose target is determined at runtime. Note that, even though methods [33] have been proposed to detect the targets of a jump table (one type of indirect branches), how to reliably detect other types of indirect branches (e.g., function pointers) is still an open research question. We do find that resolving indirect jump targets can improve the result of disassembly tools (Section 4.4.2).

2.3 Function Identification

Function identification is an important primitive, which can be used to construct other primitives, e.g., function call graph. Previous work usually leverages function signatures to detect functions. The method proposed in [27, 31] scans a binary for known function prologues and epilogues. However, this method is limited by the fact that function signatures (e.g., prologues and epilogues) could be missing. Moreover, it’s a tedious task to maintain a up-to-date signature database.

Due to these limitations, a compiler-agnostic function identification method was proposed in Nucleus [38]. The basic idea is to analyze the inter-procedural control flow graph (ICFG). To use Nucleus, the binary should have distinguished function call instructions, e.g., the call1 instruction for x86. Unfortunately, this assumption does not hold for the binary compiled using the Thumb instruction set. Specifically, the BL $1abe1$ instruction is intended for a direct function call. However, Thumb binaries reuse it for a direct branch (besides the function call) since the range of the branch target is larger than the B $1abe1$ instruction. As a result, it causes many false positives to the function detection. We observed a significant decrease of the precision value for the function boundary of binaries with the Thumb instruction set (Figure 8(b)).

2.4 Code Obfuscation

Some binaries are obfuscated. For instance, O-LLVM [45] is usually used to obfuscate programs. It supports the following obfuscation methods. Specifically, instructions substitution (sub) is used to replace standard operators with more complicated instruction sequences. The bogus control flow (bcft) changes a function’s control flow graph by adding basic blocks. The control flow flattening (f1a) uses the control flow flatten algorithm [49] to create a large number of fake control flows. In this study, we use O-LLVM to generate the obfuscated binaries and feed them to disassembly tools to evaluate the impact of code obfuscation.
we first generate the ground truth by leveraging the debugging tools. As shown in Figure 3, we first compile various types of programs, aiming to understand which compiling options are mostly used in compilers used in the wild. Considering the diversity of compilers, we use two compilers, i.e., GCC and Clang, each with three different versions. Specifically, we use GCC versions 6.5, 7.5 and 8.3 and Clang versions 7.0, 8.0 and 9.0, which cover the major compiler versions.

Different Types of Programs We use three types of programs in our study. They include the widely used SPEC CPU2006, binaries in AOSP and OpenWRT. The latter two represent the binaries for mobile systems and IoT devices. Specifically, we compile the SPEC CPU2006 using both Clang and GCC compilers with two optimization levels (O1 and O2), two instruction sets (ARM and Thumb) and two CPU target architectures (-march) with ARMv5 and ARMv7. In total, we get 608 binaries, i.e., 19 benchmark programs × 2 optimization levels × 2 compilers × (3 compiler versions + 1 specific CPU architecture with latest version of compilers) = 608 binaries.

Considering the popularity of IoT and mobile systems, we build the latest Android Open Source Project (AOSP version 9) and extract daemon binaries (127 in total) and libraries (667 in total). Also, we build the latest stable version of OpenWRT (version 18.06). There are 12 different target boards that support the ARM architecture. In total, we get 246 binaries.

Compilers We use two compilers, i.e., GCC and Clang, each with three different versions. Specifically, we use GCC versions 6.5, 7.5 and 8.3 and Clang versions 7.0, 8.0 and 9.0, which cover the major compilers used in the wild.

Compiling Options As mentioned in Section 2.1, different compiling options (e.g., with or without -mthumb) would result in completely different binaries. Considering the diversity of binaries, we aim to understand which compiling options are mostly used in the real world. Thus, we divide ARM binaries into three types, according to systems they are used.

1. Type-I: Embedded OSes They are used in resource-constrained ARM devices, mainly the ARM Cortex-M processor families with low computational power. We select FreeRTOS v10.1.1 [23], the most popular real time operating system, and Mbed OS (version 5) [3], the open-source embedded operating system designed for IoT. There are several projects in FreeRTOS, each supports a different development board (or device). We cross-compile all the projects that support the ARM architecture. For the Mbed OS, we compile all the targets that support the ARM architecture.

2. Type-II: Linux Kernel Linux kernel has been used on ARM devices widely. Such devices include mobile devices and ARM servers. For mobile devices, we use the most popular operating system Android, and build the kernel (version 4.4.169) for Android 9.0 (code name: Pie). We also build the kernel for Debian (version 9.6.0), one of the most popular Linux distributions for desktop computers and servers. We download and cross-compile the kernel (version 4.9.144) from Debian’s official repository.

3. Type-III: User-level Programs We also use user-level programs, including daemons, libraries used on mobile devices, desktop computers and servers. Specifically, we build user-level programs from Buildroot [6], Android Open Source Project (AOSP, version 9.0.8) [1], and the popular Debian packages. They are representative programs for low-end embedded systems, mobile devices, and ARM desktop/servers. In particular, Buildroot is commonly used in low-end embedded systems, including routers, IP cameras and etc. We compile all binaries targeting ARM development boards. For Debian packages, we use the top five mostly installed packages, i.e., libpam-modules, libattr1, libpmem0g, zlib1g, and libsnutils, ranked by the Debian popularity contest [9].

Table 2 shows the result of compiling options for Type-I, Type-II and Type-III binaries. We find that the Thumb instruction set is mostly used in Type-I binaries, and O2 and Os are commonly used optimization levels. Due to this observation, we compile the benchmark programs for the evaluation to both ARM and Thumb instruction sets with O2 and Os optimization levels to reflect real situations of ARM binaries in the wild.

Obfuscation To evaluate the impact of obfuscation, we use the O-LLVM [45], an open-source obfuscator, to compile the SPEC CPU2006. O-LLVM supports three different obfuscation methods, i.e., instruction substitution(sub), bogus control flow graph (bcf) and control flow flattening (fla). We apply each obfuscation method to each program, and then combine three methods together. Since the bogus control flow graph (bcf) consumes too much time (more than 2 hours) when applying it to C++ programs, we do not apply this method to C++ programs.

Summary In total, we get 1,896 binaries including 248 obfuscated ones. We believe this dataset is representative to demonstrate the diversity of compilers, compiling options, target architectures and types of devices.
Table 2: The compiling options for Type-I, Type-II and Type-III binaries. The third column shows the number of total object files (.o files), and the last two columns show the number of object files with the Thumb instruction set and optimization levels.

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th># of Objects</th>
<th>Boards/Targets</th>
<th>Thumb</th>
<th>Optimization Levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type-I</td>
<td>Mbed</td>
<td>39,183</td>
<td>61</td>
<td>39,183</td>
<td>['0a': 39,183]</td>
</tr>
<tr>
<td></td>
<td>FreeRTOS</td>
<td>87</td>
<td>9</td>
<td>7+15</td>
<td>['0a': 24, '02': 32]</td>
</tr>
<tr>
<td>Type-II</td>
<td>Linux Kernel (Android)</td>
<td>1,361</td>
<td>1</td>
<td>0</td>
<td>['0a': 1, '02': 1, 291, '00': 1]</td>
</tr>
<tr>
<td></td>
<td>Linux Kernel (Debian)</td>
<td>1,860</td>
<td>1</td>
<td>0</td>
<td>['03': 1, '02': 1, 788, '00': 1, '0a': 1]</td>
</tr>
<tr>
<td>Type-III</td>
<td>AOSP</td>
<td>3,384</td>
<td>1</td>
<td>2,875</td>
<td>['0a': 2, 787, '02': 299, '00': 27, '03': 69]</td>
</tr>
<tr>
<td></td>
<td>Buildroot</td>
<td>188,387</td>
<td>103</td>
<td>1,677</td>
<td>['02': 188, 387]</td>
</tr>
<tr>
<td></td>
<td>Debian Packages</td>
<td>339</td>
<td>5</td>
<td>0</td>
<td>['02': 305, '03': 34]</td>
</tr>
</tbody>
</table>

Table 3: A summary of the evaluated disassembly tools and the APIs or commands that are used to retrieve the results. NA: not applicable.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Commercial Tool</th>
<th>Instruction Boundary</th>
<th>Function Boundary</th>
</tr>
</thead>
<tbody>
<tr>
<td>angr</td>
<td>8.19.4.5</td>
<td>N</td>
<td>graph.node.instruction addr</td>
<td>address%2 == 1</td>
</tr>
<tr>
<td>BAP</td>
<td>1.6.0</td>
<td>N</td>
<td>-dasm</td>
<td>NA</td>
</tr>
<tr>
<td>Objdump</td>
<td>2.30</td>
<td>N</td>
<td>-dasm</td>
<td>NA</td>
</tr>
<tr>
<td>Ghidra</td>
<td>9.0.4</td>
<td>N</td>
<td>getFirstInstruction()</td>
<td>getRegister('TMode')</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>getInstructionAfter()</td>
<td>getFunctionAfter()</td>
</tr>
<tr>
<td>Radare2</td>
<td>3.6.0</td>
<td>N</td>
<td>pD</td>
<td>NA</td>
</tr>
<tr>
<td>Binary Ninja</td>
<td>1.1.149</td>
<td>Y</td>
<td>binaryview.instructions()</td>
<td>functions.arch</td>
</tr>
<tr>
<td>Hopper</td>
<td>4.5.13</td>
<td>Y</td>
<td>getTypeAtAddress()</td>
<td>getArchitecture()</td>
</tr>
<tr>
<td>IIDA Pro</td>
<td>7.3</td>
<td>Y</td>
<td>isCode()</td>
<td>getEntryPoint()</td>
</tr>
</tbody>
</table>

3.2 Determine Disassembly Primitives

In this work, we consider the instruction and function boundary as fundamental primitives (Table 1). Other ones (e.g., direct control flow graph) could be built upon them.

Instruction Boundary The instruction boundary refers to the start offset of an instruction, as well as the correct instruction set (ARM or Thumb). The purpose is two-fold. First, it is used to distinguish between code and inline data. Inline data is commonly used in ARM binaries, e.g., for the PC-relative addressing. This is different from x86 binaries, which do not contain inline data [25] except for the jump tables of binaries compiled by Visual Studio. Second, it is used to distinguish between ARM and Thumb instruction sets. This is challenging since the instruction set is partially determined by the target address of an (indirect) branch instruction, which is hard to be obtained by static analysis.

Function Boundary The function boundary refers to the start offset of a function. Function boundary recognition is a necessary primitive to construct the call graph, which is critical to the whole program analysis.

Interestingly, tools that could precisely recover instruction and function boundary do not scale. For instance, certain tools cannot finish the analysis within two CPU hours for some binaries. That means these tools are not scalable to large and complex binaries. Table 4 shows the number of binaries (the Timeout column) that cannot be analyzed within two CPU hours.

3.3 Generate Ground Truth

After determining the primitives, we need to get the ground truth. However, even with debugging symbols, it is not straightforward to directly get the result. We describe our approaches as follows.

Instruction Boundary We use mapping symbols [2] in the binaries to get the information of the instruction boundaries. Mapping symbols are generated by compilers to identify inline transitions between code and data, as well as ARM and Thumb instruction sets. There are three types of mapping symbols, including:

- $a$: Start of a region of code containing ARM instructions.
- $c$: Start of a region of code containing Thumb instructions.
- $d$: Start of a region of data.

For instance, the mapping symbol "00001043c $c" denotes that the offset 0x0001043c in the binary is code (not inline data), with the Thumb instruction set.

However, mapping symbols only include the start address of the code and data regions without indicating the offset and the instruction set of each instruction in the region. To deal with this issue, we use Capstone [7] to retrieve the offset of each instruction. It works well since we have the instruction set (ARM/Thumb) information of each code region to help Capstone disassemble the code region.

Note that, the mapping symbol is an architecture-specific extension of the ARM ELF file. It may not exist in other architectures. By leveraging it, our system can detect the instruction boundary with a sound and complete result. Previous work can only detect 98% of the ground truth and requires a manual verification [25].

Function Boundary We leverage DWARF [39], a debugging file format to retrieve the function boundary. DWARF uses the data structure named Debugging Information Entry (DIE) to describe each variable, type, and function, etc. Each DIE has a tag (i.e., DW_TAG_subprogram) for function and each function has a key (i.e., DW_AT_low_pc) to represent the function start address.

We extract the DW_TAG_subprogram and DW_TAG_subprogram from the DWARF of each binary to get the ground truth.

3.4 Extract the Result

We evaluate eight state-of-the-art ARM disassembly tools, including five noncommercial ones, i.e., angr [60], BAP [31], Objdump [18],
Table 4: The result of whole data set. Invalid means the tool cannot identify any instructions or functions. Timeout means the tool cannot finish the analysis in two hours (CPU time). Exception means the tool raises exceptions during the analysis. Segfault means the tool triggers a segment fault during the analysis.

<table>
<thead>
<tr>
<th>Tool Type</th>
<th>Tool</th>
<th>Instruction Boundary</th>
<th>Function Boundary</th>
<th># of Timeout</th>
<th># of Exception</th>
<th># of Segfault</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Precision</td>
<td>Recall</td>
<td>F1 Score</td>
<td># of Invalid</td>
<td>Precision</td>
</tr>
<tr>
<td>Noncommercial</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.886</td>
<td>0.795</td>
<td>0.830</td>
<td>1</td>
<td>0.404</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.565</td>
<td>0.277</td>
<td>0.309</td>
<td>11</td>
<td>0.533</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.702</td>
<td>0.750</td>
<td>0.722</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.934</td>
<td>0.828</td>
<td>0.973</td>
<td>0</td>
<td>0.835</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.749</td>
<td>0.633</td>
<td>0.768</td>
<td>0</td>
<td>0.906</td>
</tr>
<tr>
<td>Commercial</td>
<td></td>
<td>0.984</td>
<td>0.987</td>
<td>0.990</td>
<td>0</td>
<td>0.906</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.971</td>
<td>0.986</td>
<td>0.978</td>
<td>0</td>
<td>0.825</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.994</td>
<td>0.970</td>
<td>0.978</td>
<td>5</td>
<td>0.914</td>
</tr>
</tbody>
</table>

Table 4 shows the overall result. The recall, precision and F1 score are computed in the granularity of macro-averaging. A tool may not be able to detect any instruction or function for a given binary; We mark such cases with the flag Invalid. We also set a threshold (two CPU hours in our study) for each tool to analyze a binary. This is because if a tool cannot finish the analysis in two hours, then it is not scaled to analyze a large number of binaries. We count the number of binaries that cannot be analyzed in two CPU hours with the flag Timeout. We also count the number of binaries that trigger an exception or a segment fault for each tool. We mark them with the flag Exception and Segfault, respectively.

4 EVALUATION

As discussed in Section 3, we build 1,896 binaries (including 248 obfuscated ones) to evaluate eight disassembly tools. We address the following research questions in Section 4.2, Section 4.3, Section 4.4 and Section 4.5 respectively.

- RQ1: What is the accuracy of disassembly tools towards the whole data set?
- RQ2: What are the factors that affect the results of disassembly tools, and what are the reasons?
- RQ3: Do different types and options of tools have different results?
- RQ4: How efficient are these disassembly tools?

4.1 Evaluation Metrics

We use precision and recall to measure the accuracy (or effectiveness) of a tool. The definition of these two metrics is in equation 1.

\[
\text{precision} = \frac{tp}{tp + fp} \quad \text{recall} = \frac{tp}{tp + fn} \tag{1}
\]

In the equation, we use \( tp \), \( fp \), \( fn \) to denote true positives, false positives and false negatives. Recall measures the ratio of true positives to the ground truth. A disassembler with high false negatives may have low recall. Precision measures the ratio of true positives to the result of a tool. A disassembler with high false positives may have low precision.

Considering the importance of both recall and precision, we also compute the F1 score according to equation 2. F1 score can reflect the overall accuracy of a tool.

\[
\text{F1 score} = \frac{2 \times \text{recall} \times \text{precision}}{\text{recall} + \text{precision}} \tag{2}
\]

4.2 Overall Results (RQ1)

Table 4 shows the overall result. The recall, precision and F1 score are computed in the granularity of macro-averaging. A tool may not be able to detect any instruction or function for a given binary; We mark such cases with the flag Invalid. We also set a threshold (two CPU hours in our study) for each tool to analyze a binary. This

Ghidra [11], Radare2 [22], and three commercial ones, i.e., Binary Ninja [5], Hopper [12] and IDA Pro [13]. Each tool has different ways to extract the instruction and function boundary. We carefully read the manual of each tool and write a script to extract the result. Table 3 lists the summary of each tool, including the version, APIs

and commands used to retrieve the result.

4.2 Overall Results (RQ1)

Table 4 shows the overall result. The recall, precision and F1 score are computed in the granularity of macro-averaging. A tool may not be able to detect any instruction or function for a given binary; We mark such cases with the flag Invalid. We also set a threshold (two CPU hours in our study) for each tool to analyze a binary. This is because if a tool cannot finish the analysis in two hours, then it is not scaled to analyze a large number of binaries. We count the number of binaries that cannot be analyzed in two CPU hours with the flag Timeout. We also count the number of binaries that trigger an exception or a segment fault for each tool. We mark them with the flag Exception and Segfault, respectively.

Note that, a tool may have different options when performing the analysis. For instance, angr provides an option to disable or enable the resolution of indirect jumps. We use the default option for each tool to calculate the overall result and leave the evaluation of the impact of different options in Section 4.4.2.

**Instruction Boundary** IDA Pro has the highest precision value, while Hopper owns the highest recall value. Both of them have the highest F1 scores and are commercial tools. Moreover, these two are robust, since they do not raise any exceptions or generate any segment faults during the analysis. Among all the tools, BAP does not perform very well on both the instruction boundary and the function boundary. This is due to the insufficient support of the Thumb instruction set. Besides, BAP does not disassemble instructions that are out of the range of recognized functions. That means if a function cannot be detected, then all instructions inside that function will be ignored. This is the reason why the recall of the instruction boundary is rather low. For other tools, the reason for the lower precision and recall mainly comes from two different reasons. One is the challenges raised by mixed ARM and Thumb instruction sets, and the other is the inline data.

**Function Boundary** IDA Pro still has the highest precision while Hopper has the highest recall. In terms of F1 score, IDA Pro has the highest value. It means that the function boundary is correlated with the instruction boundary. BAP mainly uses signatures learnt from a set of binaries to detect the function boundary. Due to the imprecise function signatures, functions with no representative signatures cannot be detected by BAP. As for Radare2, the recall is relatively low compared with other tools. That is because Radare2 has a very strict policy on detecting functions. Users can use the command aaa to explore more functions by searching for the function patterns.

**Robustness and Scalability** We find some noncommercial tools are not robust. For instance, more than 600 binaries triggered either an exception or a segment fault of angr. For the 262 binaries that triggered a segment fault, 160 of them are binaries compiled from the SPEC CPU2006, and 87.5% (140/160) of them are compiled using the Thumb instruction set. Based on this observation, there is a great
Figure 4: False positive and false negative of BAP

Figure 5: Hopper misidentifies the instruction set.

Figure 6: Radare2 cannot identify the inline data. It tries to disassemble data as code, even the disassembled instructions are invalid.

Figure 7: Ghidra performs differently when instruction set is different of this error. Specifically, the basic block (0x710bc) is indirectly reached from other basic blocks, thus it is hard for the tool to determine the right instruction set. Remember that, the instruction set is determined by the last bit of the target address.

4.3 Different Factors (RQ2)

Our data set consists of binaries that are built using different compilers, compiling options, and target architectures. Some of them are even obfuscated. They represent the diversity of existing binaries in the wild. In the following, we further explore multiple factors that affect the accuracy of disassembly tools.

4.3.1 Instruction Sets. ARM and Thumb instruction sets are widely used in real-world binaries. To evaluate the impact of instruction sets, we divide binaries into two categories. The first one contains binaries compiled with the flag -mthumb, which use the Thumb instruction set. We call them Thumb set binaries. The other one is compiled without the flag -mthumb. By default, compilers use ARM instruction set. We call them ARM set binaries.

Figure 8 shows the evaluation result. The solid line and dotted line in the figure are used to denote the precision and recall, respectively. The x-axis shows the name of tools and the y-axis represents the average value of recall and precision for all the binaries. Note that, this format also applies to Figures 9, 10, 11, 12, 13 and 14.
such an example. The instructions at the offset 0x00070078 and 0x000505bc are same (BL), which represent a function call. Both function calls refer to the same callee according to the source code. However, since the instruction set is different, Ghidra misinterprets that the callee function in the Thumb instruction set is a non-return function, thus it completely ignores the code after that offset (0x000505bc). Several similar cases are observed for the tool. This is the reason why the recall is relatively low for Thumb set binaries of Ghidra.

Third, the result of the function boundary correlates with the instruction boundary. That’s because these two primitives have a strong connection with each other. If the instruction boundary cannot be recognized precisely, it will greatly affect the recognition of the function boundary, and vice versa.

Fourth, the result of the function boundary is worse for the Thumb instruction set. We suspect that is due to the reuse of the BL label instruction as both a function call and a direct branch for Thumb set binaries. Specifically, the BL label (BLX label) instructions are used to directly invoke a function. For the ARM instruction set, compilers use instructions, e.g., B label, for a direct branch. However, for the Thumb instruction set, the range of the B label is limited (±2KB for 16-bit Thumb) [4]. Compilers tend to reuse the BL label for a direct branch (range is ±4MB for 16-bit Thumb), which is same with a function call. This confuses the disassembly tools, which misinterpret direct branches as function calls. This raises high false positives to identify the function boundary and results in a low precision. Due to this, the proposed method to identify function boundary without relying on function signatures in Nucleus [38] is also ineffective, since it assumes the function call instruction could be identified. The initial result of applying this tool to binaries with the Thumb instruction set show that both the precision and recall are below 0.12.

**Summary** The Thumb instruction set does bring serious challenges to disassembly tools.

4.3.2 **Optimization Levels.** As shown in the Section 3.1, optimization levels 02 and 0a are mostly used ones. They represent the optimization for performance and size, respectively. To evaluate the impact of optimization levels, we divide binaries into two categories. One contains binaries compiled with the 02 flag, while the other one contains binaries compiled with the 0a flag.

Figure 9 shows the result. Surprisingly, there is no significant differences between these two flags in terms of both recall and precision. To verify the conclusion that optimization level does not bring significant difference, we conducted an extra hypothesis test. We compute the F1 scores of the binaries in the two categories and compare the differences between every pair of binaries (i.e. one compiled with the 02 flag while the other one compiled with the flag 0a). We then randomly picked 40 samples and conducted t-test on the samples. Table 5 shows the result. We noticed that different optimization levels do not bring significant differences on all the eight tools. The maximum differences in terms of F1 score is only 0.065.

We further explore the potential reason. It turns out that the 0a flag enables all the optimization methods introduced in the 02 flag. Besides, it includes the ones to reduce binary size [8, 10], e.g., reducing the padding size and alignment. These ones have little impacts for the disassembly tool to identify the instruction and function boundary.

**Summary** Optimization levels (02 and 0a) do not bring significant differences.

4.3.3 **Compilers.** GCC and Clang are two popular compilers. To evaluate the impact of compilers, we build binaries (the SPEC CPU2006) with both GCC and Clang. Figure 10 shows the evaluation result.

For the instruction boundary, most tools do not have obvious differences between binaries built with different compilers except BAP, which will be explained later. However, for the function boundary, Radare2 and BAP are sensitive to binaries built with different compilers. For Radare2, the precision of the function boundary for binaries built with GCC is higher than the binaries built with Clang. BAP has a higher precision of the function boundary for binaries compiled with GCC. That is because BAP has a better collection of function signatures for binaries compiled with GCC than the ones compiled with Clang. Remember that, BAP does not disassemble the instructions that are not in the detected functions. Thus, the precision of the instruction boundary will also be higher for binaries compiled with GCC.
Summary: Compilers do not affect most of the tools, except Radare2 and BAP, mainly due to the function identification method used by them.

4.3.4 Target CPU Architectures. ARM has multiple architectures, e.g., ARMv7 and ARMv5. Each architecture has different hardware features. For instance, the 16-bit Thumb instruction (Thumb-1) is available from ARMv4, while the 32-bit Thumb-2 instructions are available from ARMv6. Thus, if the binary is built for different architectures, instructions generated by the compilers will be different.

To evaluate the impact of binaries built with different CPU architectures, we use the binaries compiled for ARMv7 (march=armv7-a) and AVRMv5 (march=armv5t). Figure 11 shows the result. We find that disassembly tools perform better for binaries with the ARMv7 architecture, in terms of the precision of function boundary. This is because the Thumb-2 instructions are supported in the ARMv7 architecture, where the B label instruction has a much larger jump range ($\pm 16$MB) than the original one ($\pm 2$KB in the Thumb-1 instruction set) [4]. Compilers tend to use the B label instruction for the direct branch, instead of reusing the BL label instruction that is usually for the direct function call (Section 4.3.1). Thus, disassembly tools can distinguish the function call instruction with the direct branch instruction, and identify the function boundary more precisely.

Summary: For the ARMv7 CPU architecture, compilers use B 1abel instruction for a direct branch, instead of reusing the BL 1abel instruction. This helps the disassembly tools distinguish the direct branch instruction with the function call instruction, leading to a better precision value of identifying the function boundary.

4.3.5 System Types. ARM binaries exist in different types of systems. In our work, we also evaluate the impact of different types of binaries. In particular, we use the binaries built from the OpenWRT [19] (Linux based embedded systems used for routers, IP cameras and etc.) and the Android open source project (AOSP version 9), respectively. The result is shown in Figure 12.

In general, the result for binaries of OpenWRT is better than the AOSP binaries. We further compared the binaries and found that most of the binaries (80%) in Android are compiled using the Thumb instruction set, while there are no binaries in OpenWRT compiled using the Thumb instruction set. As explained in previous sections, disassembly tools perform worse for Thumb binaries.

Summary: System types affect the result. This is due to the instruction set used in the binaries.

4.3.6 Obfuscation. To evaluate the impact of obfuscation to disassembly tools, we use O-LLVM [45], an open-source obfuscator, to compile the SPEC CPU2006. O-LLVM supports the following obfuscation methods. Specifically, instruction substitution (sub) is used to replace standard operators with more complicated sequences of instructions. The bogus control flow (bcf) changes a function call graph by adding basic blocks. The control flow flattening (fla) uses the control flow flatten algorithm [49] to create a large number of fake control flows.

We apply each obfuscation method to each program for building the binary, and then combine three methods together. We divide the obfuscated binaries into four groups. In the first three groups, each group contains the binaries that are obfuscated using one individual method. The last group contains the binaries that are obfuscated using all the three methods. The result is shown in Figure 13.

We observe that obfuscation does not affect the instruction boundary too much. However, the function boundary is greatly affected by the control flow flattening. This is because the control flow flattening generates a huge number of fake control flows. These fake control flows are using the BL 1abel instruction in the Thumb binaries for direct branches. These instructions confuse the disassembly tools and introduce false positives to the function boundary (Section 4.3.1).

Summary: Obfuscation introduces challenges to the disassembly tools to locate the function boundary, especially the control flow flattening. The root cause is due to the reuse of BL 1abel instruction for direct branches, which are inserted by the obfuscation tool.
We observe that enabling the indirect jump resolving will increase the precision and recall, since it can resolve more code sections that could only be reached through indirect branches. Among all the tools, Ghidra consumes most of the memory (nearly 1 GB in maximum), while Objdump consumes the least. IDA Pro is quite stable for the memory usage. It consumes only around 100MB memory for nearly 70% of the binaries.

4.5 Efficiency of the Tools (RQ4)

Efficiency is an important feature of disassembly tools. Efficient tools can handle large binaries within a reasonable time. We report the efficiency of the tools. In particular, we calculate the CPU usage, CPU times and memory consumption during the analysis. Our experiments are done in Ubuntu 18.04 with 128GB memory size and 30 core Intel(R) Xeon(R) Silver 4110 CPUs. Specifically, we use the Python library psutil [21] to extract the related information about resource consumption, and then use the function cpu_times to obtain the CPU times. We also use the function cpu_percent (interval = 1) to extract the CPU percentage. Note that this value can be bigger than 100% in case of a process running multiple threads. We use the function memory_info to obtain the memory consumption. The memory size is the Resident Set Size (rss), which is the non-swapped physical memory a process has used. The result is shown in Figure 15.

CPU Percentage Binary Ninja consumes lots of CPU resource and can reach to nearly 800% for some binaries. Ghidra ranks the second. Half of the binaries would consume 200% of the CPU usage. BAP consumes the least CPU percentage. However, according to our observation, BAP spends a lot of time to analyze the binaries due to the inefficient usage of CPU.

CPU Times Among all the tools, Ghidra consumes most CPU times compared with the other tools in nearly 80% binaries. There are no significant performance differences for angr with and without indirect jump resolving. Since the indirect jump resolving improves the result, we recommend users to enable this option during analysis. For Radare2, the option aaa needs much more CPU times compared with the option aa. However, the precision and recall of the instruction boundary do not have a significant improvement.

Memory Consumption Among all the tools, Binary Ninja consumes most of the memory (nearly 1 GB in maximum), while Objdump consumes the least. IDA Pro is quite stable for the memory usage. It consumes only around 100MB memory for nearly 70% of the binaries.
ARM-specific disassembly strategies

First, inline data is popular in ARM binaries. Previous research shows that there is few inline data in x86/x64 binaries and the jump tables are located in the .rodata section. However, inline data is very common in ARM binaries, which increases the difficulty to locate instruction boundaries. Second, there are two instruction sets, i.e., ARM and Thumb instruction sets. Detecting the right instruction set is challenging for disassembly tools. Furthermore, we noticed that most tools do not have good support on the Thumb instruction set, either with a wrongly detected instruction set or a thrown exception. For instance, angr throws exceptions and gets segment faults for several binaries with the Thumb instruction set. `Objdump` can merely identify the Thumb instruction set. Given the fact that the Thumb instruction set is popular, especially in the binaries for mobile systems, there is an urgent need to propose effective solutions. Third, since most existing works are focusing on x86 and x64 \cite{25, 27, 28, 38, 63, 64}, some ARM specific mechanisms should be proposed to deal with the instruction set switching. For instance, the hybrid disassembly technique \cite{70} could be leveraged to locate the inline data and distinguish between different instruction sets, with customizations to adapt to the ARM architecture. Besides this, disassembly tools could perform a further check on its disassembly result. In other words, they could conduct a conflict analysis to improve the result. For example, `Radare2` explicitly knows when there is an invalid instruction. In this case, it can either switch the mode, or further check whether the invalid code is actually inline data through a data reference analysis.

Mechanisms to identify the function boundary

Our result shows that there is still a large space to improve the effectiveness of detecting the function boundary. Tools usually use function signatures to identify functions. These signatures could be generated through a machine learning based method. However, the machine-learning based methods could be limited due to the incompleteness of the training data sets \cite{38}. For instance, the machine-learning based method in BAP performs worse than most of the other tools in detecting function boundaries. Furthermore, the mechanisms that work well on x86/x64 \cite{38} cannot be applied to ARM, because ARM does not have a distinguished function call instruction, which is required by the method. According to our evaluation, besides function call, BL 1abe1 is widely used in the Thumb instruction set for direct branch. Disassembly tools cannot distinguish the usage of BL 1abe1 as direct branch with direct function call, resulting in a low precision in terms of the function boundary.

We think a more effective algorithm to detect the function boundary is needed. For example, developers could use the machine-learning based mechanism to detect the function first, and then conduct a static analysis by considering the internal logic between different basic blocks to reduce the false positives and false negatives. Moreover, disassembly tools can further analyze the BL 1abe1 instruction to understand whether it’s a function call. For instance, if the target of the instruction BL 1abe1 is also reached via the B 1abe1 instruction, then the target is not the start of a function. We think a further analysis of the usage of the BL 1abe1 instruction can greatly improve the result of the function boundary.

Usability

Tools have different user interfaces and plugin infrastructures. For instance, IDA Pro, Hopper and Binary Ninja have user-friendly GUI interfaces, and provide easy-to-use Python APIs. `angr` itself does not provide GUI, and is invoked purely through a Python script. BAP has a good flexible architecture for extension. However, the supported language Ocaml has a steep learning curve, compared to the Python programming language. As for `Radare2`, it is completely different from the other tools. It just loads the binary and provides an interactive shell. Users have to leverage the shell to perform the analysis. There are many different kinds of built-in analysis phases.

We also observe that non-commercial tools suffer from the scalability and stability. For instance, BAP cannot finish the analysis on several binaries while angr will raise exceptions or get segment faults on several binaries. Furthermore, tools may have different options, which impact the usage of system resources. Users should pick the right options according to the purpose. For example, if users use the `Radare2` to disassemble the instruction (and do not care about the function boundary), they can use the option aa, which satisfies the need and is much faster than other options.

6 DISCUSSION

First, with the introduction of the ARMv8 architecture, there exist 64-bit ARM binaries, which are missed in this work. However, 32-bit
binaries are still the most popular ones. Due to the compatibility concern, new ARM architectures maintain backward compatibility with old ones. Our findings in this paper can still be applied to ARMv8 (ARMv8 supports both AArch64 for 64-bit binaries and AArch32 for 32-bit binaries) and future versions of ARM as long as ARM does not deprecate 32-bit ARM instruction set. Besides, AArch64 simplifies the task of disassembly tools. This is because 32-bit ARM has both 16-bit and 32-bit instructions and much more diverse branch instructions. As shown in our evaluation, the switching between instruction sets brings serious challenges to disassembly tools.

Second, we only evaluate eight state-of-the-art disassembly tools. However, there exist some disassemblers that are either research prototypes or not actively maintained. They are excluded from our work. Moreover, we only evaluate two fundamental disassembly primitives. We think other primitives such as direct control flow graph or direct call graph are easy to be generated if the instruction boundary and function boundary are located correctly.2

Third, the generation of the ground truth is an essential step. Fortunately, the ARM ELF format introduces mapping symbols that can help to distinguish between different instruction sets, and between code and inline data. By leveraging this information, we can generate a complete and sound result for the instruction boundary. At the same time, we could use the DWARF debugging information to extract the ground truth of the function boundary. For other primitives like control flow graph and call graph, they consist of direct jumps and indirect jumps. Direct jumps can be built based on the precise instruction boundary and function boundary, which is the reason why we do not include them in the evaluation. For the evaluation of indirect jumps, we cannot get a sound and complete ground truth even if we have the source code. This is because some jump targets can only be determined at running time. We leave the evaluation of these primitives as one of the further works.

7 RELATED WORK

The most related one is the study of x86/x64 disassembly tools [25]. However, there are several differences between the study and ours. First, ARM supports mapping symbols, which can help us to extract the precise ground truth. Second, we focus on the metrics of recall and precision when evaluating the different factors (e.g., optimization levels and instruction modes), which can reveal the tools’ limitations. Third, the conclusion of x86/x64 study cannot be applied to ARM binaries. For example, inline data is very common in ARM binaries, which is rare in x86/x64 binaries. Apart from this, there are two instruction set in ARM architecture, which require more precise analysis mechanism for tools to identify the right instruction set. Furthermore, the reuse of BL 1ab1 instruction for both function call and direct branch brings challenges to disassemblers to detect the function boundary.

Zhang et al. [70] combined the linear sweep and recursive traversal. However, their work is for x86 binaries and there is no experiment describing the accuracy of this algorithm. Ben et al. [29] proposed the idea of speculative disassembly on Thumb binaries with the assumption that binaries are not obfuscated. However, their work is not scalable to real world binaries as ARM instruction set are widely used. Though Kruegel et al. [47] proposed an algorithm on obfuscated binaries, their work does not target the ARM architecture. Bauman et al. [28] proposed the idea of superset disassembly, while Miller et al. [51] proposed the probabilistic disassembly mechanism. However, they only focus on x86/x64 binaries.

Detecting the function boundary is also a challenging research topic. Rosenblum et al. [56] use the machine learning technique to identify functions. Other works [27, 31, 44, 46, 58] extended this idea with different machine learning algorithms. However, it is rather hard to build a general model. Other tools [20, 48, 60] use heuristics or hard-coded signatures to identify the function boundary. The fundamental problem is that there exist functions that do not have the signatures or do not follow the heuristics. Qiao et al. [55] applied static analysis to detect the function boundary. However, their work only targets the x86 architecture. Dennis et al. [38] designed a new methodology on detecting function boundaries by analyzing control flow graphs. However, it assumes there is a distinguished function call instruction (e.g., the call instruction in x86/x64) in the binary. This mechanism cannot be applied to ARM due to no distinguished function call instruction in ARM binaries.

As discussed, disassembly tools have been widely used in existing works. Taegyu et al. [62] designed and implemented a rewriter for ARM binaries. It requires precise disassembly results. Directed fuzzing [30, 32] use the control flow graph to calculate distances between basic blocks. Without correct instruction boundary, it is hard to build the precise control flow graphs. Vulnerability detection by applying binary similarity techniques [50, 66, 72] relies on the accurate disassembly results. Different algorithms [52, 68, 70, 71] have been proposed to build control flow graphs. Since some applications are leveraging off-the-shelf tools to disassemble binaries, the evaluation can help improve these tools, which further improves the result of applications that depend on these tools.

8 CONCLUSION

In this paper, we conduct the first comprehensive study on the capability of eight ARM disassembly tools to locate instruction and function boundaries, using diverse ARM binaries built with different compiling options and compilers. We report our new findings, which shed light on the limitations of the state-of-the-art disassembly tools, and point out potential directions for improvements.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their comments that greatly helped improve the presentation of this paper. We also want to thank Zhi Wang for the helpful discussion. This work is supported in part by Hong Kong RGC Project (No. 152239/18E), the National Natural Science Foundation of China (NSFC) under Grant 61872438, Leading Innovative and Entrepreneur Team Introduction Program of Zhejiang (2018R01005), Zhejiang Key R&D 2019C03133, Singapore National Research Foundation, under its National Cybersecurity R&D Program (Grant Nos.: NRF2018NCR-NCR005-0001), National Satellite of Excellence in Trustworthy Software System (Grant Nos.: NRF2018NCR-NSOE003-0001), NRF Investigatorship (Grant Nos.: NRF06-2020-0022), and DARPA under agreement number FA875019C0003.

2We understand that the indirect control flow transfer is still a challenging task.
et al. 2016. Sok:


