IDAPro for IoT Malware analysis?

Sri Shaila G, Ahmad Darki, Michalis Faloutsos, Nael Abu-Ghazaleh and Manu Sridharan
Computer Science and Engineering Dept.
University of California, Riverside
{sg001@,adark001@,michalis@cs.,nael@cs.,manu@cs.}ucr.edu

Abstract

Defending against the threat of IoT malware will require new techniques and tools. An important security capability, that precedes a number of security analyses, is the ability to reverse engineer IoT malware binaries effectively. A key question is whether PC-oriented disassemblers can be effective on IoT malware, given the difference in the malware programs and the processors that support them. In this paper, we develop a systematic approach and a tool for evaluating the effectiveness of disassemblers on IoT malware binaries. The key components of the approach are: (a) we find the source code for 20 real-world malware programs, (b) we compile them to form a test set of 240 binaries using various compiler optimization options, device architectures, and considering both stripped and unstripped versions of the binaries, and (c) we establish the ground-truth for all these binaries for six disassembly accuracy metrics, such as the percentage of correctly disassembled instructions, and the accuracy of the control flow graph. Overall, we find that IDA Pro performs well for unstripped binaries with a precision and recall accuracy of over 85% for all the metrics. However, IDA Pro's performance deteriorates significantly with stripped binaries, mainly because the recall accuracy of identifying the start of functions drops to around 60% for both platforms. The results for the stripped ARM and MIPS binaries are similar to stripped x86 binaries in [2]. Interestingly, we find that most compiler optimization options, except the -O3 option for the MIPS architecture, do not cause any noticeable effect in the accuracy. We view our approach as an important capability for assessing and improving reverse engineering tools focusing on IoT malware.

1 Introduction

IoT malware is emerging as the new battleground of cybersecurity. Leaders in the technology industry like Ericsson forecast that there will be around 18 billion devices related to the Internet of Things (IoT) [11]. The expansion of the scope of IoT devices has redefined communication and information transfer between users and smart devices. While this trend continues to offer a new level of connectivity and convenience, it also poses a serious threat to the security and the privacy of users and the stored data in the devices. Recent attacks on IoT devices such as Mirai [5] and Moose [6] highlight the need to defend these devices. On the other hand, IoT malware developers release the source code to the public as it has been the case for Mirai [4], and *Lightaidra* [3]. Black hat hackers make use of such source code to create malware that targets IoT devices to engage in malicious activities [19]. Further, the software eco-system for IoT devices is quite heterogeneous and not as mature with respect to security [10].

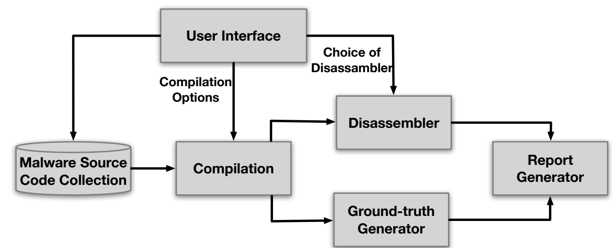


Figure 1: The visual overview of our approach for evaluating the effectiveness of a disassembler on IoT malware.

In this work, we address the question of the effectiveness of binary disassemblers for analyzing IoT malware. These tools are critical first steps in reverse engineering malware binaries, which is necessary to enable a number of analyses of their security. Thus, answering this question will shed light on the larger issue of the effectiveness of the existing PC-focused defense mechanisms for IoT malware. We focus on IDA Pro [13], which is a state of the art disassembler for binary analysis [2]. To elaborate, we seek to assess the reverse engineering capabilities of IDA Pro by using six popularly used disassembly accuracy metrics, for a set of ARM or MIPS IoT malware binaries. These metrics assess different aspects of the accuracy of IDA Pro and are important for the correct structure of the disassembled code, as well as the construction of its control flow graph. The metrics include the percentage of correctly disassembled and identified instructions, function starts, function parameter counts, basic

blocks, control flow graph (CFG), and call graph accuracy. We also seek to investigate the effect that various compilation options have on the disassembly performance.

Key challenge. Ideally, the disassembly accuracy or the effectiveness of a dissasembler should be measured by establishing the structural and semantic similarity between the source code and the assembly code of the binary. However, the syntax differences between the two languages is a challenging task that requires extensive manual effort.

To the best of our knowledge, there has not been any previous studies of the performance of disassemblers on IoT malware. The closest related work by Andriesse et al. [2] evaluates the performance of 9 disassemblers for the `x86` architecture (PC-based) on benign binaries. Other related work can be grouped into the following categories: (a) static analysis of malware [7, 14]; (b) disassemblers in malware analysis: using disassemblers in analyzing the malware structure [15, 17]; (c) IoT binary analysis: analyzing code similarities of IoT firmware to identify known vulnerabilities [12, 21]; and (d) IoT malware analysis: analyzing the behavior of IoT malware [5, 19, 10]. We discuss related work further in Section 6.

As our key contribution, we develop a systematic and comprehensive approach and the resultant tool for assessing the effectiveness of disassemblers on IoT malware binaries. The novelty of our work can be summarized as follows:

a. We evaluate the performance of disassemblers for the ARM and MIPS architectures: We evaluate the performance of disassemblers for the `ARM` and `MIPS` architectures on malware programs. Each of these architectures have their own instruction sets, hence new non-trivial tools are needed. By contrast, previous work [2] focused on the `x86` architecture (PC-based) on benign software, the SPEC CPU2006 benchmark, which consists of the standard *libc* binaries.

b. Our approach is fully automated: In contrast to previous effort [2], our approach obtains the ground-truth for 100% of the code bytes automatically, and thus, can be used in large-scale studies.

The key components of our approach are outlined below:

Step 1: Source code and compilation. We collect source code for 20 malware programs and compile it into 240 binaries comprising of 200 unstripped and 40 stripped binaries with different settings: (i) different compilation options, (ii) two different CPU architectures, ARM and MIPS, and (iii) both stripped and unstripped versions of the binaries. This process is represented by the Malware Source Code Collection and the *Compilation* modules in Figure 1. None of the malware programs are obfuscated.

Step 2: Establishing the ground-truth. We obtain the ground-truth for the binaries for all the disassembly metrics, which is non-trivial despite having the source code. We determine the ground-truth for a comprehensive set of metrics: *Instructions*, *Basic Blocks*, *Function-Start Addresses*, *Number of Non-Zero Function Parameters*, *Control Flow Graph* (*CFG*), and *Call Graph*, which we define later. This process is represented by the *Ground-Truth Generator* module in Figure 1.

Step 3: Applying the disassembler. We apply the under investigation disassembler to the binaries and extract the results for all the metrics. This step is represented by the *Disassembler* module in Figure 1.

Step 4: Evaluation. We compare the results of the disassembler with the ground-truth to generate the evaluation report. This step is represented by the *Report Generator* module in Figure 1.

Results. We demonstrate our technique by using it evaluate the widely-used IDA Pro [13]. Our findings can be summarized as follows:

a. IDA Pro performs well on unstripped binaries. We find that IDA Pro does well across all metrics of interest for both architectures, and all compilation options for unstripped binaries. The percentage accuracy for all metrics is over 85%. Specifically, we find that ARM binaries seem to lend themselves to more accurate reconstruction compared to MIPS binaries, although the difference is relatively small.

b. Stripped binaries challenge the performance of IDA Pro. We find that IDA Pro fails to correctly identify a significant fraction of *Function-Start Addresses* for stripped binaries for both architectures. Also, IDA Pro misses 37.7% and 39.6% of the functions in stripped binaries for ARM and MIPS respectively. We conjecture that in the absence of the symbol table, the function recognition algorithm of IDA Pro has a hard time determining the beginning and end of functions.

c. Compilation options have limited effect. We studied the effect of different compiler options on disassembler effectiveness for unstripped binaries. Interestingly, we found that the compilation optimization options do not have any noticeable effect on any of the six metrics for the unstripped binaries for both architectures, and this is particularly true for ARM. For MIPS, there are a few exceptions, where some non-trivial change is observed. For example, the `-O3` compiler option, and to a lesser extent `-O2`, cause a small drop in the detection of *Function-Start Addresses* for the unstripped MIPS binaries.

We argue that our work provides an important building block for developing tools and methodologies for reverse engineering malware. This work has led to the following tangible outcomes: (a) a set of IoT malware source codes, (b) ground truth, and (c) a usable platform that can help accelerate static analysis and disassemblers in the IoT malware space.

2 Methodology

In this section, we present our approach for evaluating the effectiveness of binary disassemblers on IoT malware. Checking the correctness of disassembled code requires a good understanding about all instructions in the architecture sets and

about how disassemblers work. The following sections will discuss the disassembly metrics we have used for the evaluation, and implementation details of the evaluation which include the workflow of our technique. Binary analysts understand and analyze binaries by viewing the assembly level instructions, basic blocks, control flow graphs, and call graphs. Each address in the code section usually contains one instruction. A basic block is comprised of a straight line code sequence with no branches in except to the entry and no branches out except at the exit. A CFG shows how the basic blocks are connected to each other. It is defined as a graph showing all the possible paths that might be traversed through a program during its execution. In our work, we consider the intra-procedural CFG, which is a graph showing how all the basic blocks within a function are connected to each other to form all the possible paths. Call graphs represent the relationships between functions in a program. Each node represents a function in the program and each edge represents a caller function that calls a callee function.

2.1 Disassembly metrics

In our evaluation, we use 6 frequently used metrics which provide a comprehensive view of the capability of the disassembler. A malware analyst uses features whose accuracy are measured by these metrics to describe the binary and to reconstruct the malware behavior. The same metrics have been used in prior reverse engineering studies [2].

We provide a description of the metrics below:

1. **Correctly identified instructions (CI):** the percentage of correctly disassembled instructions. Disassembled instructions at an address location in which the ground-truth contains an instruction are considered as correctly disassembled instructions.
2. **Correctly identified function starts (CFS):** the percentage of correctly identified function start addresses. Function starts addresses which are matched with the ground-truth are considered as correctly identified functions starts. Otherwise they are considered as incorrect function starts (**IFS**).
3. **Correctly identified function parameters (CFP):** the percentage of functions with non-zero parameters for which the number of function parameters was correctly identified by the disassembler when compared to the function parameter number supplied by DWARF.
4. **Correctly identified Basic Blocks (CIBB):** the percentage of basic blocks with the correct start and end address containing all instructions within that address ranges as shown by the ground-truth.
5. **CFG Accuracy (CFGA):** the percentage of basic blocks found within a function that have the correct start and end address with the correct number of successor blocks that also have the correct start and end address as shown in ground-truth. We only consider a basic block as having the correct connections in the CFG if it satisfies all these conditions. Otherwise we mark that basic block as having incorrect connections. Here, we are referring to inter procedural control flow graph.
6. **Call Graph Accuracy (CGA):** the percentage of correctly identified instructions that invoke another function which are found under the correct function as shown in the ground-truth.

2.2 Overview of our Approach

Our approach consists of several steps, which we outlined in the introduction. Here, we discuss each step in more detail.

Step 1: Source code and compilation. We were able to collect the source code of 20 IoT malware programs,from two resources (14 from the first source and 6 from the second): a) live websites such as `Github.com` and `Pastebin.com` where developers released the code for research purposes,[1] and b) source codes shared as archives of files with instructions on Black hat hackers' websites. These source codes were randomly selected from the sites and all source codes were written in the C language.

Having the source code, we compile them in different ways to evaluate the disassembler. Specifically, we consider the effects that different architectures and compiler optimization options have on the accuracy of IDA Pro's binary disassembly as we explain below.

a. Compiler options. We study the effects of compiler options like `-O{0,1,2,3,s}` where each optimize the binary differently: compilation time, execution time, or the size of the binary. We also study the impact of `stripped` binaries where any debugging information such as function names is removed. We used the `GCC v5.5.0` cross compiler to generate 240 binaries from malware source codes.

Our technique can be applied to generate the ground-truth to evaluate all possible compiler options. However, in view of the page limit restrictions, we only present the results of the evaluation of IDA Pro for selected compiler options.

b. Target architectures. We generate binaries for both the ARM (Version 5, Little Endian) and MIPS (R3000, Little Endian) architectures, which are the dominant architectures used in embedded and IoT devices. In the following sections, we will refer to these architectures as ARM and MIPS respectively. Note that both these architecture are popular among IoT devices, and they differ in instruction sets from PC-based (x86 & x86_64) architectures.

Step 2: Establishing the ground-truth. Here, we establish the ground-truth for all the binaries that we have generated. In this work we use a combination of *DWARF v3* [8] and *Capstone v3.0.4* [18] to create the ground-truth.

[1] https://github.com/ifding/iot-malware

DWARF is used to extract the debugging information for each variable, type and procedure from the source code while Capstone translates binary bytes into assembly instructions for the respective architectures.

We use the `-g3` compiler option in order to obtain the DWARF debugging information. DWARF provides a mapping between source level code and the assembly instructions in the binary. Stripped binaries do not contain this information. The ground truth information of the corresponding unstripped binary is used to evaluate the stripped binaries.

We used *Capstone* to linearly disassemble instructions starting from an initial set of known addresses provided by the DWARF mapping. This set comprises of the entry point and function start address.The linear disassembling continues from one instruction to the next in a linear ascending order until the end of the function is reached.

This seemingly straight forward solution contains a challenge. Architectures like ARM contain inline data [16] that needs to be identified and labelled as data in our ground-truth.

We overcome this problem by studying how the inlined data is used by assembly code. Inlined data is used by the assembly code by using load operations which use pc-relative addressing modes. Since the pc register always points to the instruction address of the next instruction, we can calculate the addresses that contain inlined data when these load instructions are disassembled. This approach yields 100% ground-truth for the code bytes in the test binaries making it suitable for large scale fully automated analysis.

We use the details and the information obtained about each instruction from Capstone to generate the ground-truth for all the metrics.

Step 3: Applying the disassembler. We apply the disassembler on all our binaries, and extract the values for each metric in our evaluation. Here, we evaluate IDA Pro 6.8 [13], as it is the most commonly used disassembler by security analysts. We have used `IDAPython` API and created scripts for each architecture to collect the information about all of the metrics for each of the understudy malware binaries. The `IDAPython` API contains functions that allows us to extract all the functions found in the binary, the basic blocks found in the functions,the instructions found in each basic block, and the functions that are called from within each function. We ran these scripts in IDA Pro's default mode to extract the values for each metric for our evaluation.

Step 4: Evaluation. The tables which summarizes the results of our findings are found in the next page. The tables show the percentage accuracy for each of the six disassembly metrics. We have computed the accuracy in terms of precision and recall for most of the metrics. The precision refers to the percentage of correctly identified metrics instances among all the instances identified by IDA Pro while the recall refers to the fraction of correctly identified metrics instances among all the metrics instances identified in the ground-truth.

3 Our Study

In this section, we present the evaluation result for IDA Pro 6.8; our techniques can be used for evaluating other disassemblers in a similar fashion.

In this study we used 20 malware source codes to generate a total of 240 binaries for both MIPS and ARM CPU architecture. Our evaluation process consists of two parts that evaluate the effect of a) compiler options and b) stripped binaries on IDA Pro disassembler tool.

1. The effect of compiler options. We assess the accuracy of IDA Pro for each of the disassembly metrics for the five compilation options: `-O0`, `-O1`,`-O2`,`-O3` and `-Os`. We use 200 binaries which includes 100 binaries for MIPS and the other 100 for ARM architecture.

2. The effect of binary stripping. This part evaluates the accuracy of the disassembler tool in retrieving the disassembly metrics for the stripped binaries. In this work we focus on stripped binaries created using `-O3` optimization option. We chose this option because it is the most commonly used option by malware authors in Makefiles. In this analysis we generated 40 stripped binaries which consists of 20 binaries for ARM and 20 for MIPS.

Since the precision for the *CI* and the *CFP* is 100% in all the options, this information is omitted in the result tables. IDA Pro incorrectly identifies functions in two ways. It may "miss" a function start or it may identify "multiple" functions within one function. The first case adversely affects recall and the second case affects precision for the *CFS* accuracy. Summing the precision and the "*extra*" percentages and recall with "*missing*" will result in 100%. In short, the percentages under the missing and the extra functions give the percentages of functions whose actual start addresses are missed and incorrectly added by IDA Pro.

3.1 The Effect of Compiler Options

Table 1 shows the effect of compilation optimization options on the accuracy of IDA Pro in disassembling ARM and MIPS binaries. We highlight some of the results and explain the observed discrepancies.

A. Performance of IDA Pro on ARM binaries. For the ARM architecture, IDA Pro is able to find all the disassembled instructions for all of the 5 different compiler optimization options. IDA Pro shows near perfect recall for *CFS* for all binaries. This shows that IDA Pro manages to identify almost all the functions in the ground-truth (zeroes in the "missing" row). On the other hand, the precision for all options falls slightly below perfect since in most cases, IDA Pro also finds an extra function for it incorrectly splits a function, `divsi3`, an integer library routine from the standard *libc* function, into two separate functions. This accounts for the percentage values in the row titled "extra", representing the additional functions that were added incorrectly.

CI	Correctly identified instructions	*CIBB*	Correctly identified Basic Blocks
CFS	Correctly identified function starts	*CFGA*	Control Flow Graph Accuracy
IFS	Incorrectly identified function starts	*CGA*	Call Graph Accuracy
CFP	Correctly identified function parameters		

		ARM					**MIPS**				
		`-O0`	`-O1`	`-O2`	`-O3`	`-Os`	`-O0`	`-O1`	`-O2`	`-O3`	`-Os`
CI	**Recall**	100	100	99.9	99.9	100	98.4	97.8	99.6	99.5	98.8
CFS	**Precision**	99.7	99.6	99.2	99.6	99.4	100	100	98.5	94.5	100
	Recall	100	100	100	100	99.9	99.9	100	100	100	99.9
IFS	**Missing**	0	0	0	0	0.05	0.06	0	0	0	0.06
	Extra	0.3	0.4	0.8	0.4	0.6	0	0	1.5	5.4	0
CFP	**Recall**	92.5	97.2	97	96.1	97.4	0	0	0	0	0
CIBB	**Precision**	99.8	100	99.9	99.9	99.9	85.3	89.0	98.9	98.5	90.6
	Recall	99.9	100	99.9	99.9	99.9	87.3	89.0	98.8	98.5	90.6
CFGA	**Precision**	99.6	99.7	99.6	99.7	99.6	99.9	99.9	98.8	94.7	99.9
	Recall	99.7	99.7	99.6	99.8	99.6	99.3	99.9	98.8	94.7	99.2
CGA	**Precision**	100	99.9	100	99.9	100	100	100	98.7	98.1	100
	Recall	100	99.9	100	99.9	100	98.9	100	98.7	98.1	98.9

Table 1: ARM & MIPS architecture: The effects of compiler optimization options over our 20 malware source codes.

		ARM		**MIPS**	
		Unstripped	**Stripped**	**Unstripped**	**Stripped**
CI	**Recall**	99.9	99.9	99.5	99.5
CFS	**Precision**	99.5	99.5	94.5	91.2
	Recall	100	62.2	100	60.3
IFS	**Missing**	0	37.7	0	39.6
	Extra	0.4	0.5	5.5	8.8
CFP	**Params**	96.1	0	0	0
CIBB	**Precisions**	99.9	99.9	98.5	97.8
	Recall	99.9	99.9	98.5	97.1
CFGA	**Precision**	99.7	99.1	94.7	89.0
	Recall	99.8	85.0	94.7	78.5
CGA	**Precision**	99.9	99.4	98.1	92.0
	Recall	99.9	88.2	98.1	89.5

Table 2: The effect of stripping: for both ARM and MIPS architectures and using optimization `-O3`

We observed an interesting phenomenon while investigating the discrepancies of IDA Pro regarding *CFP* which shows the percentage of correctly identified number of parameters from the functions with non-empty parameter. We observe that IDA Pro, identifies 3 parameters for the `main()` function. These parameters are reported as: `int argc`, `const char **argv`, and `const char ** envp`. However, only the first 2 are found in the source code as well as the ground-truth data. This is why we observe a not perfect accuracy for the *CFP* for the `-O1`, `-O2`, `-O3`, and `-Os` options. Specifically, IDA Pro did not find any parameters for some functions with non-zero parameters for one of the binaries with the `-Os` compilation option. This caused the parameter accuracy for the `-Os` compilation to be worse than the other compilation options.

The *CFGA* accuracy is over 99%. Upon investigation we found that the incorrect splitting of functions, like the `divi3` function mentioned above, accounts for the imperfect precision and recall. We also noticed that a few *CIBB* were being split into two. This further created isolated blocks in the CFG. The precision and the recall for the *CFGA* is very close to perfect across all the 5 compilation options.

B. Performance of IDA Pro on MIPS binaries. For the MIPS architecture, we noticed a slight drop in the recall for the *CI* primitive compared to the ARM binaries. Upon investigation, we attributed this to the fact that some instructions were not correctly identified by IDA Pro. Further analysis revealed that most of the missing instructions are “add” instructions.

The *CFS* metric shows perfect precision for `-O0`, `-O1`, and `-Os` for the MIPS binaries and it has close to perfect recall for all 5 compilation options. We see that 0.06% of the func-

tions in the ground-truth missed by IDA Pro for the -O0 and -Os compiling options, while 1.5% and 5.4% of the functions reported for -O2 and -O3 compiling options are additional functions that are incorrectly found by IDA Pro. Since *intra-procedural* CFG accuracy is directly dependent on correct identification of function starts, CFG accuracy drops as the *CFS* accuracy drops. IDA Pro generated the most number of additional functions for MIPS binaries compiled with the -O3 compiler option. IDA Pro is not able to retrieve any parameters for the MIPS architecture.

We consider a block as being matched, if the start address and the end address and the number of instructions within the block match a basic block in the ground-truth. Hence, in general, the matched *CIBB* accuracy suffers as the percentage of missed instructions increases. The number and the distribution of the missed instructions also affect the percentage of matched *CIBB*. If the missing instructions are spread out across a larger number of *CIBB*, then the number of matched *CIBB* will be reduced. This causes the variations in the accuracy of matched *CIBB* across the compiler options for the MIPS binaries. Precision and recall of matched *CIBB* is the lowest, when the binary is compiled with -O0, and highest for -O2 and -O3 option.

For CFG accuracy, we only take into account the start and end address of each of the *Basic Blocks* and the start and end addresses of the successive connected *CIBB*. Hence, the presence of missing instructions does not affect the CFG accuracy. Since CFG accuracy decreases as the number of incorrectly identified additional functions increases, the CFG accuracy of the binary with -O3 compilation is the lowest with 94.7% precision and 94.7% recall. The CFG accuracy for MIPS binaries compiled with the other options for both the recall and precision is about 98.2% to 99.9%.

The precision and recall for fucntion start also affect the *CGA* accuracy. MIPS binaries compiled with the -O1 option have perfect precision and recall because all the functions were identified correctly. The other binaries have less than perfect accuracy, because some of the call instructions were wrongly categorized as belonging to another function.

3.2 The Effect of Binary Stripping

We compare the results between stripped and unstripped binaries in our study in Table 2. IDA Pro finds all the instructions for both the stripped and the unstripped binaries for the ARM architecture. The number of missed instructions remains about the same for the MIPS malware binaries.

However, stripped binaries make the detection of *CFS* much harder: the number of missed functions increases significantly to 37.7% for stripped ARM binaries and to 39.6% for stripped MIPS binaries. Upon investigation, we found two reasons for these failures: (a) IDA Pro cannot associate some instructions to any function, and usually displays the instructions in red color as , and (b) parts of a function may also be erroneously attributed to another function.

IDA Pro identifies a higher percentage of extra functions erroneously in stripped MIPS binaries compared to stripped ARM binaries. This causes the precision and the recall for the *CFGA* metric for the MIPS stripped binary to be lowered to 85.0% and 78.5% respectively. Incorrect function identification could be attributed to both: (a) failing to report functions that exist in the ground-truth, and (b) reporting extra functions due to erraneous splitting of a function by IDA Pro erroneously.

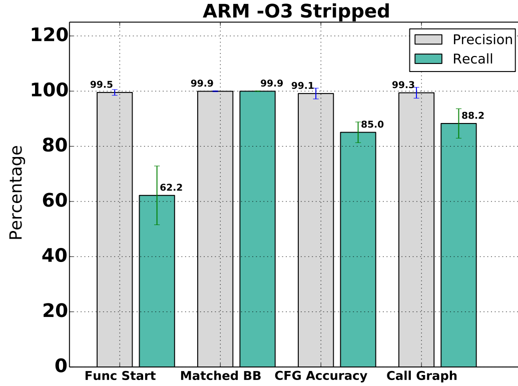
Interestingly, even though the errors due to function misses is high, around 37.7% and 39.6% for the ARM and MIPS stripped binaries, their adverse effect on the CFG accuracy is relatively small. Upon investigation, we found that IDA Pro tends to miss smaller functions with few basic blocks. Precision and recall for this *CFGA* metric is noticeably higher for the ARM (vs. the MIPS) unstripped binaries due to the much higher accuracy in function start identification. The percentage of missed functions in unstripped binaries for both platforms is 0%.

For unstripped binaries compiled with the -O3 option, the percentage of extra functions incorrectly identified by IDA Pro is higher for the MIPS binaries with 5.4%, compared to the ARM binaries with only 0.4% additional functions. The matched basic block percentage precision is over 99% for the ARM binaries, while these percentages for the MIPS binaries is slightly lower. We attribute this to the larger fraction of missing instructions in the MIPS binaries compared to the ARM binaries. -O0 misses the most number of instructions while -O2 and -O3 miss the least.

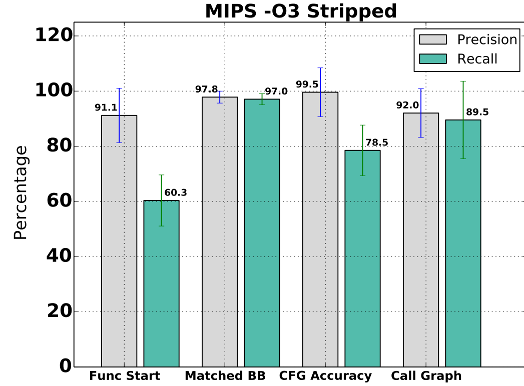
We observe that the *CGA* precision for the unstripped ARM binary is very close to 100%. The call graph precision and recall for the MIPS unstripped binary falls slightly to 98.12%. We attribute the errors to the larger fraction of the call instructions, which were categorized to belong to the ”extra” functions found by IDA Pro.

In stripped binaries, the *CGA* precision in ARM binaries stands at 99.4% because the number of ”extra” functions found by IDA Pro is much lesser for the ARM architecture than compared to the MIPS architecture. The *CGA* precision for the stripped MIPS binaries falls to 92.0% due the larger fraction of call instructions that were categorised as being part of the ”extra” functions. The recall for the *CGA* of both the ARM and MIPS stripped binaries falls to to 88.2% and 89.5%, due to the increased percentage of missing functions.

Stripping has limited effect on the *CGA* precision for ARM binaries because the percentage of incorrectly identified additional functions found is very low, around 0.5% for both stripped and unstripped binaries. There is noticeable fall in precision for *CGA* from 98.1% to 92.0% when MIPS binaries are stripped. This is because stripping in MIPS binaries leads to an increase in the percentage of incorrectly identified additional functions from 5.5% to 8.8%. Stripping leads a significant reduction in recall for *CGA* in both architectures because it causes a significant increase in the



(a) Precision and Recall for stripped ARM binaries



(b) Precision and Recall for stripped MIPS binaries

Figure 2: ARM & MIPS stripped binaries: Mean and variance for the metrics in our result.

percentage of missed functions, 37.7% for the ARM architecture and 39.6% for MIPS architecture when compared to unstripped binaries. When call instructions are incorrectly placed under the wrong functions, the precision and recall of *CGA* will be adversely affected.

Figure 2 shows the mean and the variance in the precision and recall for each of the metrics for each of the unstripped binaries. It shows that Ida Pro works consistently well on all the stripped binaries for both architectures for *CIBB*. We noticed a high variation of about 10% for recall for *CFS* for both architectures. The variation in recall for *CFGA* and *CGA* for stripped MIPS binaries is 9% and 14% respectively. In contrast, these values for stripped ARM binaries are 3.7% and 5.3%. A possible reason for this observation could be that functions with lesser number of blocks and direct calls are consistently missed across all the stripped ARM binaries

The recall for the *CFS* is 62.2% and 60.3% and has a high variance of 10.6% and 9.2% for ARM and MIPS respectively.

4 Discussion

Our findings show that IDA Pro does well for unstripped ARM and MIPS binaries for the various compiler options with greater than 90% and 85% accuracy across all metrics. We ignore *CFP* in MIPS binaries because IDA Pro is unable to retrieve parameters for this architecture.

Our results for the ARM and MIPS stripped binaries are similar to the results from [2] for stripped x86 binaries for the *CI* and *CIBB*. These metrics have precision and recall of more than 97%.

IDA Pro misses 37.7% and 39.6% of the *CFS* in the ARM and MIPS stripped binaries respectively. These percentages are slightly higher that the percentages shown in the previous work [2] for the stripped x86 binaries which stands around 35%. Our recall for the *CGA* metric is 88.2% and 89.5% for the stripped ARM and MIPS binaries respectively. In contrast, [2] reports perfect recall for stripped x86 binaries. We could not compare our results for the *CFGA*, because the previous work has considered inter-procedural CFG. We have considered intra-proecedural CFG, since that is the default output from IDA Pro. IDA Pro could not retrieve the parameters for unstripped ARM and MIPS binaries.

Disassemblers usually identify function starts and ends by scanning the binary for known series of instructions that usually form the start and end of functions [1]. The large portion of function start misses by IDA Pro for both the ARM and MIPS binaries suggest that the function prologue and epilogue signature databases are missing a some commonly found function prologues and epilogues.

The decision to use IDA Pro for stripped binaries depends on the metrics and the accuracy required by the analyst for their work. If we desire a recall exceeding 85%, then we cannot rely on the *CFS* and *CFGA* generated by IDA Pro. Hence, tools that rely on these metrics will suffer from limited accuracy as well.

5 Limitations and Future Work

In this work, we have used the source code of 20 IoT malware programs, which we compiled using various compiler options. We have focused on malware for this project because many recent research efforts [12, 21, 5, 19, 10] use binary disassemblers like IDA Pro to analyze IoT malware. We believe that our work will give malware researchers a better idea about the level of accuracy that they can expect from IDA Pro. In this work, we evaluate IDA Pro 6.8 because this was the version that we had experience with and was available to us. Here, we discuss the limitations of our work and future improvements.

Benign Binaries. We did not assess the performance of IDA Pro on benign binaries. It would be interesting to see if the performance of IDA Pro varies between benign and malicious software.

Other Compilers. We have used the `GCC v5.5.0` compiler to compile the malware source codes for the experiments. We have used this GCC version because its the most widely used and commonly supported GCC version by many

tools [20]. It would be interesting to assess the effect of different compilers on the performance of IDA Pro.

Other Platforms. We have compiled our malware source codes into ARM (Version 5, Little Endian) and MIPS (R3000, Little Endian) binaries and used these binaries in our test suite. Other commonly used architectures for IoT malware include x86-64, PowerPC and Motorola 68000 [9], and we plan to evaluate these platforms in the future.

Evasion. Malware authors employ evasive techniques like obfuscation to hinder analysts from reading and understanding their code. Obfuscated code and packing techniques are also applied to confuse disassemblers. The malware programs that we have used are unobfuscated.

6 Related Work

To the best of our knowledge, no previous studies have been done on the performance of disassemblers on IoT malware. A recent and extensive study [2] assess the effectiveness of disassemblers on PC-based architecture and with benign software in contrast to the IoT architecture and malware source code that we use here. Specifically, the study evaluates the performance of 9 commonly used disassemblers for x86 architecture and using software from the SPEC CPU2006 benchmark. Overall, they found IDAPro to be the best disassembler. They found that IDAPro does well for correctly identifying *Instructions*, *Basic Blocks* and *Call Graph*. However, it does poorly in identifying *Function-Start Addresses* and *Number of Non-Zero Function Parameters*.

We highlight some of the related work based on the following categories.

Static analysis of malware. These works statically analyze malware binaries to extract features from *CFGA* and frequently found code bytes to detect malware binaries [7, 14].

Disassemblers in malware analysis. Several efforts use disassemblers in analyzing the malware structure like call graphs [15]. These studies use disassemblers for malware classification. However, and they do not evaluate the performance of the disassemblers that they have used in their work. These works serve as motivation for our work which is to understand the effectiveness of disassemblers which are crucial tools in malware detection and classification techniques.

IoT firmware analysis: detecting vulnerability. These efforts analyze the binary code of IoT firmware to identify known bugs and vulnerabilities by extracting features from *CFG* so that they can be fixed in a timely manner [12, 21].

IoT malware analysis. There have been several studies analyzing the behavior of IoT malware [5, 19, 17, 10]. The studies focus mostly on the behavior, and the spread patterns of IoT malware and using static and dynamic analysis.

7 Conclusion

We develop a comprehensive and systematic method and the resultant tool for evaluating the effectiveness of disassemblers on IoT malware binaries. We apply our tool on IDAPro [13], a widely-used disassembler in the binary analysis research. We assess the performance of the tool on six disassembly metrics, which capture how well we can recover instructions, basic blocks, and the control flow graph. We also explore a wide range of compilation options and consider two target architectures, ARM and MIPS.

Overall, we find that IDAPro works quite well for unstripped binaries across all primitives of interest (e.g. $\geq$ 85% recovery accuracy) and for both architectures. However, stripped binaries seem to present significant challenges: IDA Pro does not perform as well with stripped binaries (e.g. function-start identification drops to 60% for both architectures. Interestingly, we find that the compilation options (-*Ox*) have limited effect on the accuracy of IDA Pro.

We view our approach as an important capability for assessing and improving reverse engineering tools focusing on malware. In addition, the malware source code repository, the ground-truth and our software tools and extensions can hopefully accelerate the research in this space.

8 Acknowledgements

This work was supported by UC Lab Fees grant LFR-18-548554. All opinions and statements reported here represent those of the authors.

References

[1] D. Andriesse, A. Slowinska, and H. Bos. Compiler-agnostic function detection in binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 177–189, April 2017.

[2] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 583–600, Austin, TX, 2016. USENIX Association.

[3] Kishore Angrishi. Turning internet of things (iot) into internet of vulnerabilities (iov): Iot botnets. *arXiv preprint arXiv:1702.03681*, 2017.

[4] Anna-senpai. [free] worlds largest net:mirai botnet, client, echo loader, cnc source code release. *https://hackforums.net/showthread.php?tid=5420472.*, 2016.

[5] Manos Antonakakis et al. Understanding the mirai botnet. In *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[6] Olivier Bilodeau and Thomas Dupuy. Dissecting Linux/Moose The Analysis of a Linux Router-based Worm Hungry for Social Networks. (May), 2015.

[7] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. Technical report, Wisconsin Univ-Madison Dept of Computer Sciences, 2006.

[8] DWARF Standards Committees. The dwarf debugging standard, 2017.

[9] E. Cozzi, M. Graziano, Y. Fratantonio, and D. Balzarotti. Understanding linux malware. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 161–175, May 2018.

[10] Ahmad Darki, Chun-Yu Chuang, Michalis Faloutsos, Zhiyun Qian, and Heng Yin. Rare: A systematic augmented router emulation for malware analysis. In *International Conference on Passive and Active Network Measurement*, pages 60–72. Springer, 2018.

[11] Ericsson. November 2018, the connected future - internet of things forecast, May 23 2018.

[12] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 480–491, New York, NY, USA, 2016. ACM.

[13] SA Hex-Rays. Ida pro disassembler, 2008.

[14] Hyang-Ah Kim and Brad Karp. Autograph: Toward automated, distributed worm signature detection. In *USENIX security symposium*, volume 286. San Diego, CA, 2004.

[15] Joris Kinable and Orestis Kostakis. Malware classification based on call graph clustering. *Journal in computer virology*, 2011.

[16] Kenneth Miller, Yonghwi Kwon, Zhuo Zhang Yi Sun, Xiangyu Zhang, and Zhiqiang Lin. Probabilistic disassembly. In *ICSE Technical Track*, Montreal, Canada, 2019. ICSE Association.

[17] Younghee Park, Douglas Reeves, Vikram Mulukutla, and Balaji Sundaravel. Fast malware classification by automated behavioral graph matching. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, page 45. ACM, 2010.

[18] Nguyen Anh Quynh. Capstone: Next-gen disassembly framework. *Black Hat USA*, 2014.

[19] Pierre-Antoine Vervier and Yun Shen. Before toasters rise up: A view into the emerging iot threat landscape. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 556–576. Springer, 2018.

[20] Tony Theodore Martin Gerhardy Tiancheng "Timothy" Gu Boris Nagae Volker Diels-Grabsch, Mark Brand. Mxe (m cross environment). *https://mxe.cc/.*, 2007-2019.

[21] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376. ACM, 2017.