UNIVERSITY OF CALIFORNIA
RIVERSIDE

MalAnalysis: A Systematic Framework for Identifying Weaknesses in Malware Detection and Analysis Tools

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Sri Shaila G

December 2021

Dissertation Committee:

Prof. Michalis Faloutsos, Chairperson
Prof. Rajiv Gupta
Prof. Nael Abu-Ghazaleh
Prof. Manu Sridharan

The Dissertation of Sri Shaila G is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

The work presented in this dissertation would not have been possible without the help of a big number of people who have supported me and my decisions in pursuing PhD. Special thanks to my PhD advisor, Prof. Michalis Faloutsos's advice and continuous support over the past years. Thank you for giving me a second chance to complete my PhD. Many thanks for all the encouraging and inspiring words during challenging times.

I also appreciate the timely feedback and insightful comments from the other committee members, Professors Rajiv Gupta, Nael Abu-Ghazaleh and Manu Sridharan. I would also like to thank the reviewers at the conferences for reviewing and accepting my research works. My first work, titled "IDAPro for IoT Malware analysis?" was published in the 12th USENIX Workshop on Cyber Security Experimentation and Test,(CSET '19) and my second work titled "DisCo: Combining Disassemblers for Improved Performance" was published in the 24th International Symposium on Research in Attacks, Intrusions and Defenses, (RAID '21).

I would like to thank my coauthors and lab mates who have helped me to improve and proof read my publications: Ahmad Darki, Md Omar Faruk Rokon, Risul Islam, Jakapun Tachaiya and Joobin Gharibshah.

My PhD journey has been both challenging and satisfying. I want to take this opportunity to thank everybody who has been a part of it. Thank you to my family and friends for their support and encouragement during this time. This would not have been possible without you all.

ABSTRACT OF THE DISSERTATION

MalAnalysis: A Systematic Framework for Identifying Weaknesses in Malware Detection and Analysis Tools

by

Sri Shaila G

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2021
Prof. Michalis Faloutsos, Chairperson

Malware infects thousands of systems globally each day causing millions of dollars in damages. Tools like anti-malware engines and disassemblers are essential front-line tools in malware defense. Anti-malware engines are used to detect malware while disassemblers are used to analyze the malware, understand its operations, and defuse it. Our overarching goal is to identify and improve our ability to detect and understand malware and consists of three major thrusts. First, we address the problem of identifying which available disassembler gives the most accurate disassembly for malware binaries of the ARM and MIPS architecture. Surprisingly, our comprehensive and systematic evaluation revealed that disassemblers have complementary capabilities. Furthermore, it also led to a bug discovery in Ghidra. Second, we leverage the results from our evaluation, identify weaknesses in disassemblers, and we develop methods to improve disassembly accuracy. As a key novelty, we develop the first approach to combine disassemblers efficiently using an ensemble approach to improve disassembly accuracy significantly. Third, we adopt a hacker-centric approach and we stress-test the effectiveness and robustness of anti-malware engines against IoT mal-

ware. Our goal is to develop evasion techniques that: (a) minimize the required effort to modify the source code and (b) preserve the functionality of the malware. Surprisingly, we find that anti-malware engines rely significantly on string matching for detection and labelling. Leveraging this, we show that some simple techniques achieve 100% evasion rate for IoT malware binaries by applying string manipulations in the source code. This thesis is a significant contribution towards: (a) assessing existing and developing new capabilities in disassembling binaries, and (b) understanding how anti-malware engines detect and label malware, especially in the space of IoT malware.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Binary disassemblers and anti-malware engines are essential front-line tools in malware defense: timely and efficient detection and reverse engineering of the malware binary is critical to identify and understand malware. An incident in 2017 highlights the importance of this issue: two ransomwares, WannaCry and Petya infected over 230,000 Windows PCs across 150 countries by exploiting a vulnerability in Microsoft's implementation of the Server Message Block protocol in a span of one day. These malwares were spreading at an alarming rate of 10,000 devices per hour, affecting systems across multiple industries rendering them unusable. These attacks cost over $14 billion dollars in damages. In addition to monetary losses, these attacks also threaten the availability of critical services. For example, system failures in hospitals prevent health care providers from accessing or updating patient information. Such situations can potentially cripple the health care system [137, 138]. In the event of these attacks, malware analysts race against time to understand their mode of

propagation and operation to contain the negative effects of the attack. These analysts also update anti-malware engines such that that they can detect similar malware in future.

Binary disassemblers are also used for other purposes like malware classification tasks [71, 136, 31] and to aid in dynamic analysis [44]. Disassemblers are also used to extract features from malware binaries. Commonly extracted features include API calls, strings, control-flow-graphs, opcode frequency and byte code n-grams [134]. Malware classification techniques are then evaluated by using anti-malware engines [162]. Disassemblers are also used to extract useful configuration information so that the dynamic analysis will have a higher chance of activating the malware [67].

The emergence of malware on Internet of Things (IoT) devices is the next battle-ground for cybersecurity and requires the development of novel security methods and tools. IoT devices have started to emerge in 1999 approximately and the term IoT encompasses a vast number of devices, including industrial controllers, home sensors, printers, smart refrigerators, and devices of the power-grid infrastructure. Market analysis estimated 5.8 billion Internet-connected IoT devices in 2020 [60]. The architecture and the variability of the configuration of these devices requires new types of software and new techniques for analyzing this software, which includes malware.

The key problems that we address in this thesis are: (a) which disassembler should a malware analyst choose to get the most accurate disassembly to detect, analyze and defuse IoT malware quickly, and (b) how easy it is for IoT malware authors to evade anti-malware engines? A variety of disassembler options, both commercial and non-commercial exist. Furthermore, malware authors can choose to compile the malware source code using

various compiling configurations. Compiling malware source code using different compilers, optimization levels and different architectures leads to different binaries at the assembly code level [27, 46]. There is limited work done on evaluating disassemblers for certain architectures. The development and emergence of new disassembler tools in recent years makes older works outdated. There is also very limited previous work done on evading anti-malware engines by applying simple techniques to modify malware source code.

This thesis makes three main original contributions in the space of IoT malware analysis, and as such, we focus on: (a) MIPS and ARM architectures, and (b) malware binaries. Firstly, we conduct a systematic and comprehensive evaluation of disassemblers by using a malware dataset. The value of our analysis is supported by our identification of a bug in the NSA-supported Ghidra disassembler. Secondly, we design and implement a novel way to combine disassemblers by using machine learning techniques to improve disassembly accuracy consistently: across compilers, architectures and compilation options. Lastly, we show that anti-malware engines are brittle as they rely significantly on string matching. As a result, introducing and manipulating strings in the source-code can lead to a significant number of false negatives and false positives.

The previous works are not comprehensive enough and do not offer any conclusive results. We will provide a detailed discussion of previous efforts in each chapter. Here, we only make the following high-level comments. Most previous work has evaluated disassemblers for the MIPS architecture. Additionally, most previous works have used benign binaries including benchmarks of benign software to conduct their evaluation, and as such, we cannot assume that the disassemblers will give similar results for malware binaries. Sec-

ondly, the work that evaluated ARM binaries only considered certain optimization levels. Thirdly, no previous work has evaluated disassemblers for MIPS binaries. Finally, no previous work has gauged how easy or difficult it is for a malware author to evade detection from anti-malware engines by applying simple techniques on IoT malware source code.

In more detail, here is a quick summary of the three thrusts of this thesis.

Thrust 1: IDAPro for IoT Malware analysis? 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. 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 [10]. 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.

Thrust 2: DisCo: Combining Disassemblers for Improved Performance Which disassembler should a malware analyst choose in order to get the most accurate disassembly and be able to detect, analyze and defuse malware quickly? There is no clear answer to this question: (a) the performance of disassemblers varies across configurations, and (b) most prior work on disassemblers focuses on benign software and the x86 CPU architecture. In this work, we take a different approach and ask: why not use all the disassemblers instead of picking one? We present *DisCo*, a novel and effective approach to harness the collective capability of a group of disassemblers combining their output into an ensemble consensus. We develop and evaluate our approach using 1760 IoT malware binaries compiled with different compilers and compiler options for the ARM and MIPS architectures. First, we show that *DisCo* can combine the collective wisdom of disassemblers effectively. For example, our approach outperforms the best contributing disassembler by as much as 17.8% in the F1 score for function start identification for MIPS binaries compiled using GCC with O3 option. Second, the collective wisdom of the disassemblers can be brought back to improve each disassembler. As a proof of concept, we show that byte-level signatures identified by *DisCo* can improve the performance of Ghidra by as much as 13.6% in terms of the F1 score. Third, we quantify the effect of the architecture, the compiler, and the compiler options on the performance of disassemblers. Finally, the systematic evaluation

within our approach led to a bug discovery in Ghidra v9.1, which was acknowledged by the Ghidra team.

Thrust 3: MalEvasion: Simple string manipulations derail malware detection How effective and robust are anti-malware engines against IoT malware? In this work, we adopt a malware author-centric point of view whose goal is to avoid detection. The key question is what is a systematic "evasion" method, which will also adhere to the following requirements: (a) minimize the required effort for modifying the source code, and (b) preserve the functionality of the malware.As key contribution, we show that anti-malware engines in VirusTotal are brittle as they rely significantly on string matching. As a result, introducing and manipulating strings in the source-code can lead to false negatives and false positives. As a proof of concept, we develop *DisCo*, a systematic framework to stress-test and confuse anti-malware engines. We evaluate our framework using 1750 binaries compiled with different compilers and compiler options for the ARM and MIPS architectures and we use the 71 anti-malware engines provided by VirusTotal. First, we show that anti-malware engines are easily fooled by our simple string manipulation techniques: 100% of the malware binaries are reported as benign if we use a widely-used detection threshold. Second, we also show that we can make engines report false positives and affect their "signature" database. We can do this by adding arbitrary strings into malware programs that we submit to VirusTotal: the engines learn to use these strings as malware signatures. Using this approach, we make 6 *string-modified* benign binaries that are deemed as malware by more than 8 engines, including Avast, AVG and Fortinet. Finally, we observe that there

is no free lunch: the engines with higher recall on malware binaries are prone to false positives on string-modified benign binaries.

Overall, this thesis constitutes a significant step towards detecting and understanding malware, and the limitations of the currents methods. We envision that our work will be used by disassembler developers to test, evaluate, and improve their tools. We also believe that our work allows malware analysts to get the most accurate disassembly outputs from all available disassemblers. Finally, our identification of weaknesses and limitations will hopefully help the security community develop better methods and tools.

Chapter 2

A Systematic Approach to Evaluate Disassemblers on IoT Malware

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) [52]. 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 [15] and Moose [20] 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 [12], and *Lightaidra* [11]. Black hat hackers make

use of such source code to create malware that targets IoT devices to engage in malicious activities [154]. Further, the software eco-system for IoT devices is quite heterogeneous and not as mature with respect to security [44].

In this chapter, 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 [66], which is a state of the art disassembler for binary analysis [10]. 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 dissassembler 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 and benign binaries. The closest related work

by Andriesse et al. [10] 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 [33, 76]; (b) disassemblers in malware analysis: using disassemblers in analyzing the malware structure [77, 112]; (c) IoT binary analysis: analyzing code similarities of IoT firmware to identify known vulnerabilities [57, 158]; and (d) IoT malware analysis: analyzing the behavior of IoT malware [15, 154, 44]. We discuss related work further in the Related Work Section.

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 [10] 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 [10], our approach obtains the ground-truth for 100% of the code bytes automatically, and thus, can be used in large-scale studies.

Results. We demonstrate our technique by using it evaluate the widely-used IDA Pro [66].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.

2.1 Disassembly Metrics

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.

2.1.1 Metrics Used

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 accu-

racy 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 [10].

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 Dataset

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.

2.3 System Design and Implementation

The key components of our approach are outlined below:

Step 1: Source code and compilation.

We compiled all the source codes in our dataset 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.

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

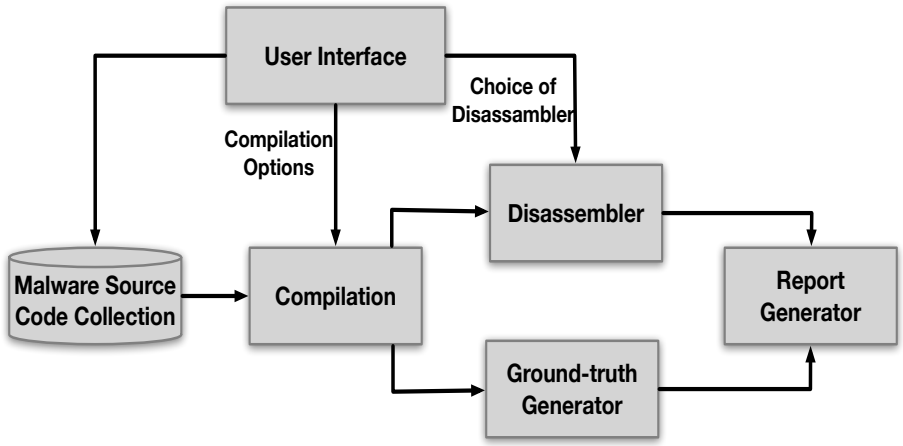


Figure 2.1: The visual overview of our approach for evaluating the effectiveness of a disassembler on IoT malware.

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* [35] and *Capstone v3.0.4* [119] to create the ground-truth. 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 [98] 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.This process is represented by the Malware Source Code Collection and the *Compilation* modules in Figure 2.1. None of the malware programs are obfuscated.

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 [66], 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. This step is represented by the *Disassembler* module in Figure 2.1.

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 2.1.

The tables in the following sections summarize 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.

2.4 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.

2.4.1 The Effect of Compiler Options

Table 2.1 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.

We observed an interesting phenomenon while investigating the discrepancies of IDA Pro regarding *CFP* which shows the percentage of correctly identified number of pa-

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

rameters 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

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

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 functions 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,

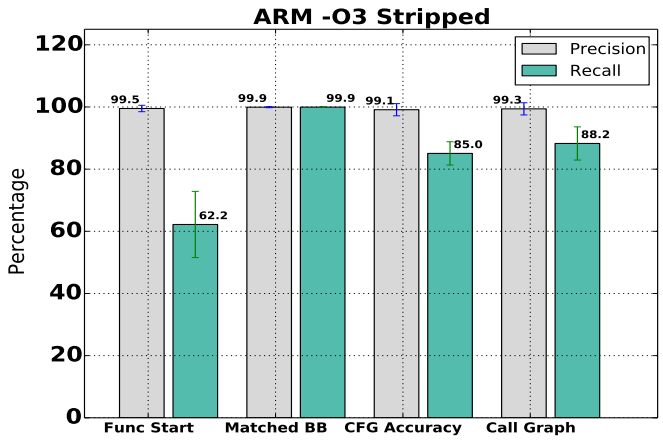
the presence of missing instructions does not affect the CFG accuracy. Since CFG accuracy decreases as the number of incorrectly identified additional functions increases, the CFG accuracy of the binary with `-O3` compilation is the lowest with 94.7% precision and 94.7% recall. The CFG accuracy for MIPS 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.

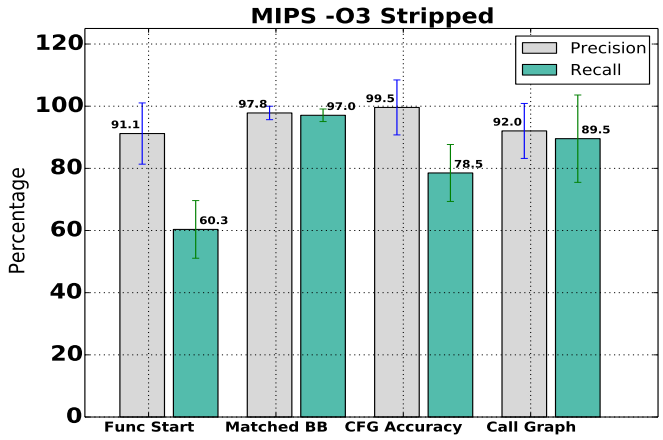
2.4.2 The Effect of Binary Stripping

We compare the results between stripped and unstripped binaries in our study in Table 2.2. 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.



(a) Precision and Recall for stripped ARM binaries



(b) Precision and Recall for stripped MIPS binaries

Figure 2.2: ARM & MIPS stripped binaries: Mean and variance for the metrics in our result.

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 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.

2.5 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 [10] 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 [10] 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, [10] 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 [8]. 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.

2.6 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 [57, 158, 15, 154, 44] 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 [50]. 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 [37], 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.

2.7 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 [10] 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 [33, 76].

Disassemblers in malware analysis. Several efforts use disassemblers in analyzing the malware structure like call graphs [77]. 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 [57, 158].

IoT malware analysis. There have been several studies analyzing the behavior of IoT malware [15, 154, 112, 44]. The studies focus mostly on the behavior, and the spread patterns of IoT malware and using static and dynamic analysis.

2.8 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 [66], 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.

Chapter 3

A Large Scale Evaluation of Disassemblers for IoT Malware

Binary disassemblers are essential front-line tools in malware defense: timely and efficient reverse engineering of the malware binary is critical. An incident in 2017 highlights the importance of this issue: two ransomwares, WannaCry and Petya infected over 230 000 Windows PCs across 150 countries by exploiting a vulnerability in Microsoft's implementation of the Server Message Block protocol in a span of one day. These attacks cost over $4 billion dollars in damages [91, 24]. The rapid spread of these malwares had malware analysts racing against time to understand their mode of propagation and operation in order to contain them.

Which disassembler should a malware analyst choose for a rapidly-spreading malware binary to get the most accurate results? This is the question that motivates our work. Here we focus on malware that targets MIPS and ARM architectures, given that such

malware has received significantly less attention. In addition, these architectures are widely used in IoT devices, which are increasingly becoming targets of choice for malware [152, 12]. Currently, there is no clear answer to the above question, for the ARM and, and even more so, the MIPS architecture. We elaborate on two contributing factors to this problem.

First, there is a plethora of disassemblers both free and commercial, but research so far has not determined a clear and consistent winner.The performance of a disassembler can vary based on the type of the malware binary, which can be created by using various: (a) compilers, (b) compiler optimization flags, and (c) target CPU architectures. These variations can lead to significant differences in the assembly code found in the resultant binary [47, 27]. Note that determining the compilation parameters for a given stripped binary is a challenging task. Despite some recent studies [10, 72], the effect of these variations on the accuracy of disassembly are not well understood, especially for the MIPS architecture.

Second, the average performance of a disassembler may not be sufficient to inform the correct answer: we need to assess its worst case performance as well. This reliability aspect of a disassembler is lost when we only report average performance, and even standard deviation does not fully capture it. The ideal disassembler should offer accurate disassembly consistently for each binary. The binary at hand may belong to the minority group of binaries for which the overall-best disassembler performs poorly. This poor performance at "crunch-time" can translate into massive financial and societal damage.

There is limited prior work for the our problem here as it focuses on: (a) malware, and (b) the MIPS and ARM architectures. In contrast, most previous work seems to focus on benign binaries and the x86, and most recently, the ARM architecture. We highlight

the two most relevant studies. The most recent work [72] evaluates several disassemblers using only benign binaries and for the ARM architecture only. Another work [10] evaluates disassemblers for the x86-64 architecture. Both studies [10, 72] endorse IDA Pro, a popular commercial disassembler,in terms of overall performance and find that accurate function start identification remains a challenge.

In this chapter, we conduct an extensive evaluation of our approach using five disassemblers on a wide spectrum of scenarios using 1760 binaries. Specifically, we consider the following **configuration options**: (a) two architectures, MIPS and ARM, (b) two different compilers, GCC and Clang, and (c) five compiler optimization levels. We compile the malware source code with various configurations and implement significant instrumentation to create the ground truth. We compare the the output of each disassembler with the ground-truth to evaluate it.

Note that we focus on the function start identification metric, which is a key disassembly metric [72]. To quantify the variability, in addition to the average performance, we consider the **5-percentile of the worst case performance (5PWC)**, which we define later. Finally, we introduce the **Relative Performance Improvement (RPI)** metric as the difference between the performance of *DisCo* for a group of disassemblers and that of the best performing individual disassembler in the group.

3.1 Dataset and Compilation Configuration

We train and evaluate *DisCo* by using a total of 1160 IoT malware binaries which were compiled from 58 IoT malware programs with various configuration options. These

programs contain a total of about 90K functions. The malware source codes were collected from Github, which hosts thousands of malware repositories [121].

Specifically, we retrieved our malware from repository *threatland/TL-BOTS*[1], which contains source files of a vast array of botnet families from 2014 to the present day. Our malware data set spans several malware families including Mirai, Gafgyt, Tsunami and Pilkah, which have been widespread in recent years [13, 135, 142, 143, 152]. Mirai, Gafgyt and Tsunami make up the majority, 89.74% of all ELF binaries that were submitted to VirusTotal between January 2015 and August 2018. [39]. Furthermore, security researchers have noticed new variants belonging to these malware families that are used to launch thousands of attacks in recent years [123, 124]. Source codes of over hundred variants of malware belonging to these families have been traced back to online repositories [39]. Hence, we believe that our dataset is representative of the malware found in the wild.

Our large scale evaluation study includes (a) various disassemblers, and (b) various compilation configurations and architectures.

The five baseline disassemblers: We consider 5 disassemblers, Angr [150], IDA Pro [66], Ghidra [108], BAP [23], and Radare2 [111] in our work. IDA Pro performed the best in previous studies and other disassemblers have been used in recent evaluation studies [72, 10].

We consider the following configurations and options.

a. Architectures: We consider two architectures, ARM version 5 and MIPS R3000. We focus on these architectures because the majority, 66.0% of the ELF malware

[1]https://github.com/threatland/TL-BOTS

binaries, belong to these architectures according to VirusTotal database [39]. Each architecture has different assembly language which requires different method and tools.

b. Compilers: We have compiled each program with two compilers: GCC version 5.5.0 and the Clang version 9.0. For the remainder of the paper, we will use GCC to refer to GCC version 5.5.0, Clang to refer to Clang version 9.0, ARM to refer to ARM version 5 and MIPS to refer to MIPS R3000.

c. Five compilation optimization levels: For each architecture and for each compiler, we have considered 5 compiler optimization levels: O0, O1, O2, O3, and Os. Each level optimizes the binary in the three-way trade-off between compilation time, execution time, and the size of the binary.

d. Stripped and unstripped binaries: We only report results on stripped binaries, because the performance of some disassemblers deteriorates significantly when binaries are stripped [59]. Furthermore, around half of all ELF malware are stripped [39].

Finally, we focus on non-obfuscated binaries, as most disassemblers are not designed to work for obfuscated binaries. A recent large scale study shows that most IoT malware is non-obfuscated and unpacked [39].

3.2 Disassembly Metrics

Note that we focus on the function start identification metric, which is a key disassembly metric [72]. To quantify the variability, we use two methods to measure performance: (a) the average, and (b) the **5-percentile of worst case performance (5PWC)** across the binaries in a testing set. The 5PWC value indicates that 5% of the binaries

perform equal to or worse than this value [94]. We argue that binary-centric performance is important for a practitioner, who, given a new binary, would like to have an estimate of its worst case performance.

3.3 Disassemblers Evaluated

We consider 5 disassemblers, Angr [150], IDA Pro [66], Ghidra [108], BAP [23], and Radare2 [111] in our work. IDA Pro performed the best in previous studies and other disassemblers have been used in recent evaluation studies [72, 10]. We consider the following configurations and options.

3.4 Ghidra Bug Discovery

In evaluating Ghidra, we ended up evaluating two versions of Ghidra, as a new version Ghidra was released during the course of our study. Specifically, Ghidra v9.1 was released in 23rd October 2019, replacing Ghidra v9.0.4. Our evaluation showed that in some cases the newer version performed worse than the older version for ARM binaries only. In figures 3.1 , we show that Ghidra v9.0.4 consistently outperforms Ghidra v9.1 for the average F1 scores for ARM binaries that were compiled with both Clang and GCC. For binaries compiled with GCC, Ghidra v9.0.4 outperforms by as much as 10.4% and 12.9% for the average and the 5PWC F1 scores. For binaries compiled with Clang, Ghidra v9.0.4 outperforms by as much as 12.2% and 19.6% for the average and the 5PWC F1 scores.

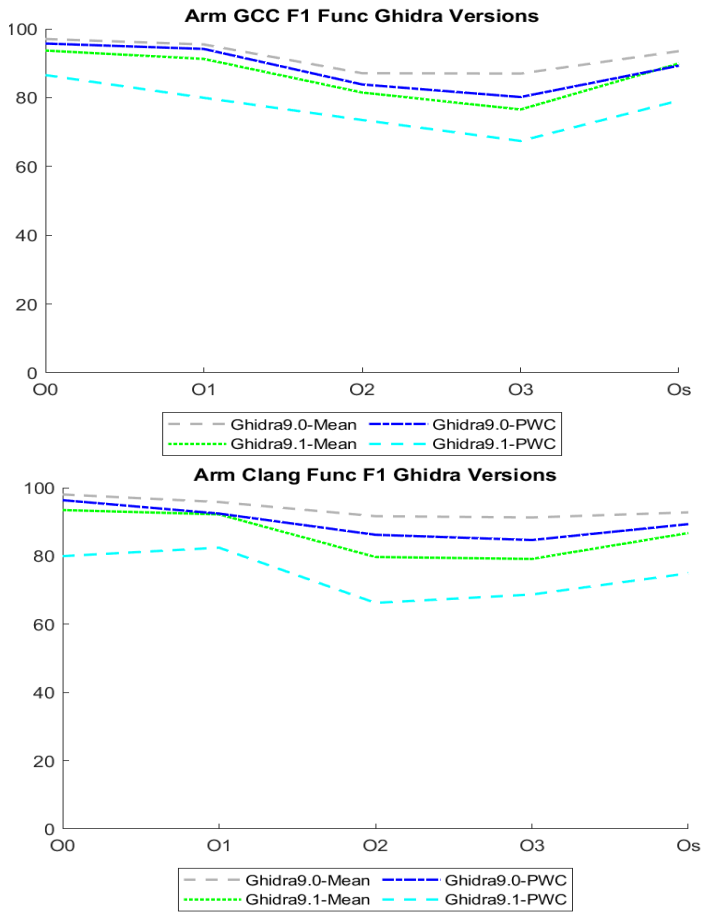


Figure 3.1: Ghidra v9.0.4 outperforms Ghidra v9.1: Mean F1 Score for ARM binaries compiled with GCC and Clang.

These figures also show that the *average* performance of Ghidra v9.1 is lower than the *worst case* performance of Ghidra v9.0.4 for most optimization levels.

Deep dive: the source of the problem. Intrigued, we wanted to understand the root cause of the problem. This involved finding the function starts where the two versions differed and then tracing what functional module of Ghidra would create this discrepancy. We traced it to a misconfiguration in the database of function start signatures in Ghidra v9.1. Specifically, we found that certain tag was attached in the function signature byte pattern for the ARM architecture. This database is named the ARM_LE_pattern.xml file and is found under the \Ghidra\Processors\ARM\data\patterns directory. This file

contains sequences of byte patterns that are known to be found at the start of functions in the ARM architectures. A new tag < *alignmark* = "0" *bits* = "3" / > was added to some of the rules, which are used to detect function starts. This tag prevented the disassembler from applying these rules, and that made it miss function starts.

We shared our discovery with the Ghidra developers, which they acknowledged. Our detailed bug report along with the suggested patch can be found under the issue section in the Ghidra repository[2] accompanied by extensive documentation. Unfortunately, this bug has not been fixed in the latest version of Ghidra, v9.2.2, which was released on 29th December 2020. We find that the database of function signatures for ARM and its performance for ARM binaries is identical to that of Ghidra v9.1. As a result, we suggest the use of Ghidra v9.0.4 for ARM binaries, while for MIPS, the newer versions can be used.

3.5 Evaluation

Overall, the evaluation results show that disassembler performance varies significantly across compilation configurations and architectures. In figures 4.4 and 4.7, we see that that (a) Angr does worse for ARM binaries than MIPS binaries, (b) Radare2 does much worse for ARM binaries compiled with Clang than ARM binaries compiled with GCC and (c) disassemblers tend to perform worse when binaries are compiled with the O2 or the O3 compilation level..

[2]https://github.com/NationalSecurityAgency/ghidra/issues/1532

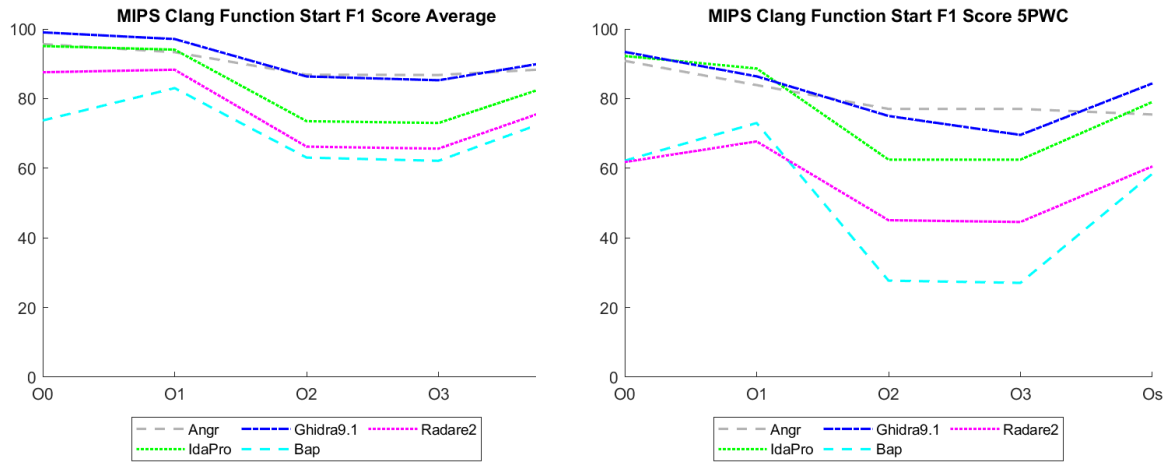


Figure 3.2: Average and 5PWC for F1 Score for Function Start Identification for MIPS binaries compiled with Clang.

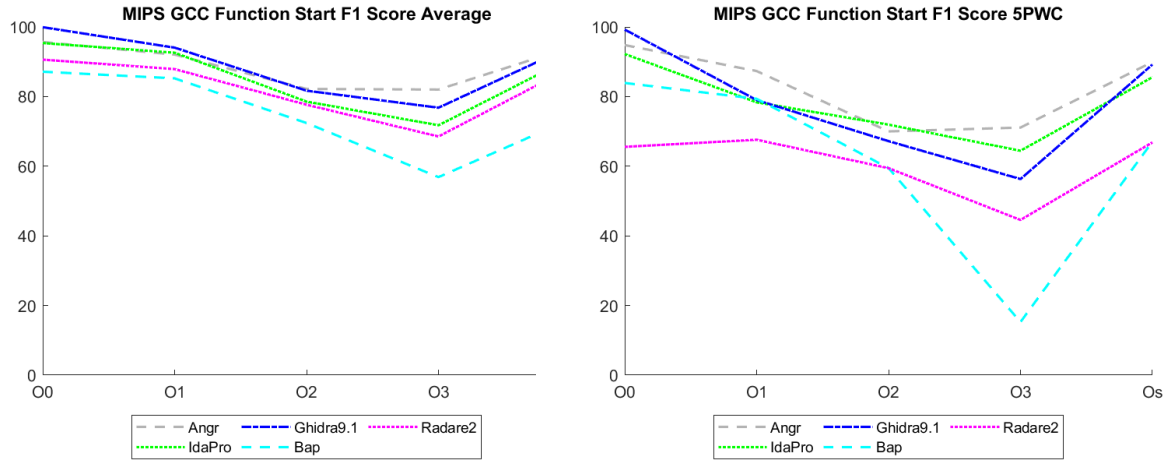


Figure 3.3: Average and 5PWC for F1 Score for Function Start Identification for MIPS binaries compiled with GCC.

We noticed that disassembler performance varies greatly even for binaries of the same configuration. In table 4.3, we observe that the differences between the average and the 5PWC F1 scores range widely between 6.3-41.5% for GCC and 9.5-34% for Clang for MIPS binaries. This suggests that the disassembly accuracy that we can expect from a disassembler can vary greatly across binaries.

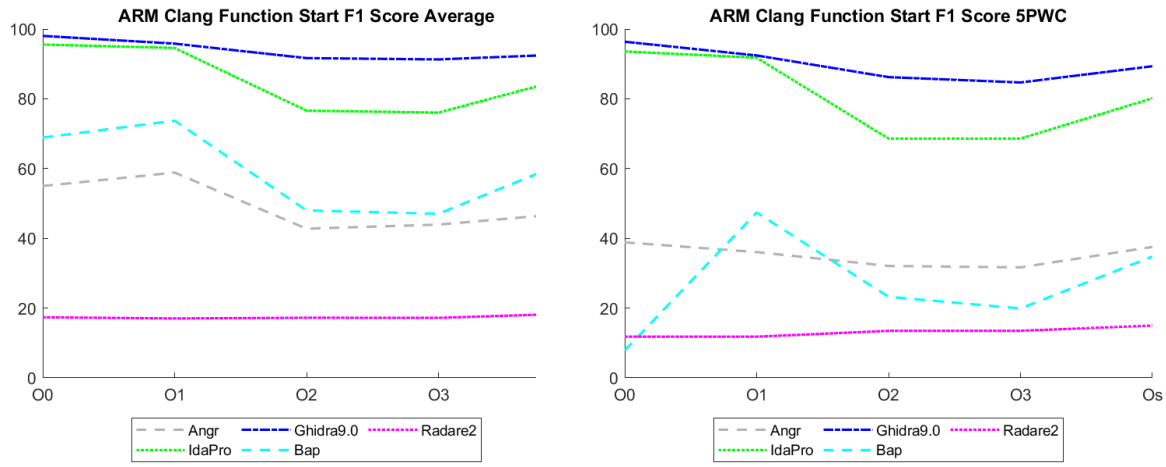


Figure 3.4: Average and 5PWC for F1 Score for Function Start Identification for ARM binaries compiled with Clang.

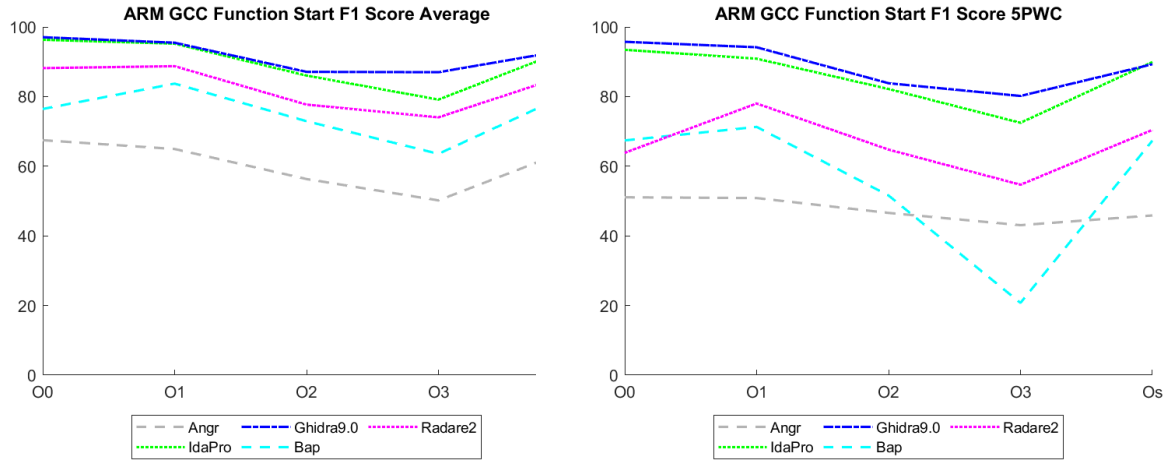


Figure 3.5: Average and 5PWC for F1 Score for Function Start Identification for ARM binaries compiled with GCC.

More interestingly, we observe that the "relative ranking" of the disassemblers can vary per binary. In fact, it is not unlikely that a lower-ranked disassembler based on average performs better for a number of binaries compared to a higher-ranked disassembler. For example, in table 4.3, we see that Ghidra outperforms IDA Pro by 5.0% in terms of the average F1 score for MIPS binaries with GCC. However, IDA Pro outperforms Ghidra by

	GCC		Clang	
	Aver.	**5PWC**	**Aver.**	**5PWC**
Angr	**82.0**	**71.5**	**86.8**	**77.3**
BAP	56.9	15.3	62.1	28.1
Ghidra	76.8	57.2	85.3	70.1
IDA Pro	71.8	65.5	73.0	63.5
Radare2	68.5	45.6	65.6	45.0

Table 3.1: **Disassembler performance varies significantly even within binaries that were compiled in the same way:** We show the average and 5PWC F1 score for function starts, *CFS*, for binaries compiled with the O3 optimization level for MIPS.

	MIPS		ARM	
	GCC	**Clang**	**GCC**	**Clang**
Angr	**82.0**	**86.8**	50.2	43.9
BAP	56.9	62.1	63.6	47.0
Ghidra	76.8	85.3	**87.0**	**91.3**
IDA Pro	71.8	73.0	79.1	76.0
Radare2	68.5	65.6	74.0	NS

Table 3.2: ***DisCo* combines disassemblers effectively:** We show the average *CFS* F1 score for binaries compiled with the O3 optimization level. NS means not supported.

8.3% for the 5PWC score. Intrigued by this, we looked at individual binaries. We found that IDA Pro outperformed Ghidra in 25.9% of the binaries in this dataset and by at least 10%. Therefore, answering the question "which is the better disassembler for a specific binary?" does not have an easy answer, even if we know the average performance.

Observation 2: Disassembler performance is affected significantly by all three factors: (a) architecture, (b) compiler, and (c) compilation levels. This observation is not surprising, but quantifying the extent of the effect is interesting. Our results in table 4.2 and 4.3 reveals several insights.

a. The effect of the architecture: Some disassemblers have better support for binaries belonging to one architecture compared to another. The more striking case is

Angr, which gives an average F1 score in the 80s for MIPS binaries, in contrast to 40s and 50s for ARM binaries and this applies to both GCC and Clang as shown in table 4.2. The effect of the architecture is interesting for BAP as it depends on the compiler: BAP with Clang does better than GCC in MIPS, but BAP with Clang does worse than GCC in ARM. The exact values are shown in table 4.2. In general, a practitioner needs to be mindful of this kind of variations for different architectures.

b. The effect of the compiler: The compiler affects the performance for some disassemblers significantly for both the average and the worst case. For example, BAP has an average F1 score of 63.6% for ARM binaries with GCC and only 47.0% for ARM binaries with Clang (see table 4.2).

Similarly, we see that Ghidra performs better for Clang for both MIPS and ARM. For MIPS, its performance increases from 76.8% for GCC to 85.3% for Clang on average, and from 57.2% for GCC to 70.1% for Clang in the worst case.

c. The effect of the compiler optimization level: Disassembler performance is affected by the compiler optimization levels used during compilation significantly. Most disassemblers tend to perform worse when binaries are compiled with the O2 or the O3 compilation level. Figures 4.4 and 4.5 illustrate this for average and 5PWC F1 score for MIPS binaries compiled with Clang and GCC. Similar trends have also been observed for ARM binaries. Figures 4.6 and 4.7 illustrate this for average and 5PWC F1 score for ARM binaries compiled with Clang and GCC.

3.6 Discussions And Future Work

Our evaluation results show that: (a) function start identification accuracy of disassemblers vary greatly even when binaries are compiled in the same way, and (b) that function start identification accuracy of disassemblers is affected by architecture, compiler and compilation levels. Hence, as a result, it is hard to pinpoint a single "best" disassembler that will perform well in all configuration scenarios.

Since a malware analyst may not know about the compilation configurations that were used to compile the binary, it would be hard for him to decide on the best disassembler to disassemble the binary at hand. We leave improving disassembly accuracy as future work.

3.7 Related Work

The most recent study [72] evaluates various disassemblers by using *benign* ARM binaries. They observe that various disassemblers offer different levels of accuracy for different types of programs. Another related work [10] evaluates the performance of disassemblers by using benign binaries for the `x86` architecture. Both works endorse IDA Pro as the best disassembler. In terms of malware binaries, a recent work [59] focuses exclusively on IDA Pro (version 6.8) and on a limited set of malware binaries. They found that malware authors tend to prefer to use the `-O{3}` options and that IDA Pro performs poorly for *CFS* for stripped binaries compiled with that option. That effort differs from our work significantly as: (a) it does not propose to combine disassemblers, and (b) it evaluates only IDA Pro in contrast to the five disassemblers that we use here.

3.8 Conclusion

Our evaluation shows that none of the disassemblers can consistently provide reliable disassembly accuracy across the malware binaries in our dataset. This highlights the urgent need to improve disassembly accuracy, which tackle next.

Chapter 4

DisCo: Combining Disassemblers for Improved Disassembly Accuracy

Binary disassemblers are essential front-line tools in malware defense: timely and efficient reverse engineering of the malware binary is critical. An incident in 2017 highlights the importance of this issue: two ransomwares, WannaCry and Petya infected over 230 000 Windows PCs across 150 countries by exploiting a vulnerability in Microsoft's implementation of the Server Message Block protocol in a span of one day. These attacks cost over $4 billion dollars in damages [91, 24]. The rapid spread of these malwares had malware analysts racing against time to understand their mode of propagation and operation in order to contain them.

Which disassembler should a malware analyst choose for a rapidly-spreading malware binary to get the most accurate results? This is the question that motivates our work. Here we focus on malware that targets MIPS and ARM architectures, given that such malware has received significantly less attention. In addition, these architectures are widely used in IoT devices, which are increasingly becoming targets of choice for malware [152, 12]. Currently, there is no clear answer to the above question, for the ARM and, and even more so, the MIPS architecture. We elaborate on two contributing factors to this problem.

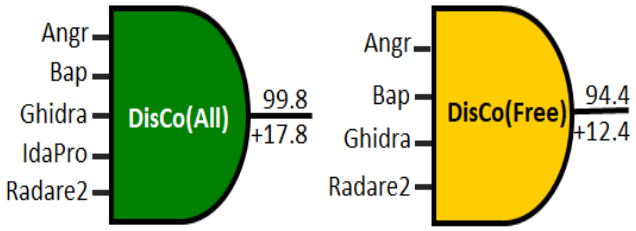


Figure 4.1: ***DisCo* effectively combines disassemblers for superior performance:** The gain in the performance over the best input can be as high as +17.8% with a combined performance 99.8% in F1 score combining five disassemblers. The results shown are for malware binaries compiled with GCC for MIPS with the O3 compilation level. The approach can work with different sets of disassemblers. We show the improvement using only freely available disassemblers (+12.4% with a total of 94.4%).

First, there is a plethora of disassemblers both free and commercial, but research so far has not determined a clear and consistent winner. The performance of a disassembler can vary based on the type of the malware binary, which can be created by using various: (a) compilers, (b) compiler optimization flags, and (c) target CPU architectures. These variations can lead to significant differences in the assembly code found in the resultant binary [47, 27]. Note that determining the compilation parameters for a given stripped binary is a challenging task. Despite some recent studies [10, 72], the effect of these variations on the accuracy of disassembly are not well understood, especially for the MIPS architecture.

Second, the average performance of a disassembler may not be sufficient to inform the correct answer: we need to assess its worst case performance as well. This reliability aspect of a disassembler is lost when we only report average performance, and even standard deviation does not fully capture it. The ideal disassembler should offer accurate disassembly consistently for each binary. Going back to our motivating example, the binary at hand may belong to the minority group of binaries for which the overall-best disassembler performs poorly. This poor performance at "crunch-time" can translate into massive financial and societal damage.

There is limited prior work for the our problem here as it focuses on: (a) malware, and (b) the MIPS and ARM architectures. In contrast, most previous work seems to focus on benign binaries and the x86, and most recently, the ARM architecture. We highlight the two most relevant studies. The most recent work [72] evaluates several disassemblers using only benign binaries and for the ARM architecture only. Another work [10] evaluates disassemblers for the x86-64 architecture. Both studies [10, 72] endorse IDA Pro, a popular commercial disassembler, in terms of overall performance and find that accurate function start identification remains a challenge. We revisit previous work in section 5.5.

In this work, we take a different approach and pose the question: *Why don't we benefit from the wisdom of all the disassemblers instead of picking one?* To this end, we present *DisCo* (**Dis**assembler **Co**mbination), a systematic approach to harness the collective capabilities of a group of disassemblers to obtain superior results. The main challenge is to ensure that the resultant model combines the strengths of the disassemblers while sidestepping the individual weaknesses. Our key contribution is an effective way to combine

disassemblers, which consists of two steps: (a) evaluating the effectiveness of each disassembler to create training data, (b) creating and training an appropriate machine learning algorithm to synthesize their individual outputs into a combined output. In the first step, we compile the malware source code with various configurations and implement significant instrumentation to create the ground truth. We compare the the output of each disassembler with the ground-truth to evaluate it and create the training data. In the second step, we use a neural network to create a stacking ensemble, which takes as input: (a) the output of each disassembler, and (b) selected data from the actual binary.

We conduct an extensive evaluation of our approach using five disassemblers on a wide spectrum of scenarios using 1760 binaries. Specifically, we consider the following **configuration options**: (a) two architectures, MIPS and ARM, (b) two different compilers, GCC and Clang, and (c) five compiler optimization levels. Note that we focus on the function start identification metric, which is a key disassembly metric [72]. To quantify the variability, in addition to the average performance, we consider the **5-percentile of the worst case performance (5PWC)**, which we define later. Finally, we introduce the **Relative Performance Improvement (RPI)** metric as the difference between the performance of *DisCo* for a group of disassemblers and that of the best performing individual disassembler in the group.

In summary, the contribution of our work can be summarized in the following key observations:

a. *DisCo* is effective in combining disassemblers. *DisCo* can combine the capabilities of different groups of disassemblers to achieve relative performance improve-

ment, RPI. In table 4.1, we show that *DisCo* achieves an RPI of up to 17.8% across various configuration options. Furthermore, there is an even larger improvement of up to 27.5% in the worst case 5PWC metric. We showcase the effectiveness of our approach visually in figure 4.1. Considering only our four non-commercial disassemblers, from table 4.1, we see that *DisCo(Free)* has an RPI of up to 12.4% for MIPS.

b. *DisCo* can be used to improve other disassemblers. We show that the collective power of the disassemblers, which *DisCo* synthesizes, can be brought back to improve each disassembler. As a proof of concept, we create, *Ghidra+*, an improved version of Ghidra, which can achieve up to 13.6% better F1 score compared to Ghidra. Our systematic evaluation of disassemblers also reveals a bug in Ghidra v9.1, which was acknowledged by the Ghidra team.

c. Configuration options affect disassembly performance significantly including their relative ranking. We find that the ranking of the best performing disassemblers varies for different configurations. For example, Angr is the best among the group for MIPS with O3 for both GCC and Clang, but it performs the lowest for the ARM architecture (see table 4.2). Furthermore, we find that the compiler optimization levels impact the performance significantly: most disassemblers do fairly well with O0 and O1, but do worse with O3 option. These observations and our results in general strongly argue in favor of the promise of a combined solution.

Open-sourcing and data sharing. Our intention is to maximize the impact of this work by enabling and encouraging the community to use our resources. We open-source our software including our code, models and signature pattern files and make our datasets

publicly available to maximize their impact.[1] We envision that *DisCo* will be used by open sourced disassembler communities to collaborate and improve their disassembly capabilities to build the next generation of disassemblers.

4.1 Dataset

We train and evaluate *DisCo* by using a total of 1760 IoT malware binaries which were compiled from 88 IoT malware programs with various configuration options. The malware source codes were collected from Github, which hosts thousands of malware repositories [121]. To avoid overfitting, we separate the training and testing datasets at the level of malware programs. Thus, we use 30 of the 88 malware programs for training and the remaining 58 programs for testing. As we are focusing on function starts, it is worth mentioning that the training set contained about 54K functions and the testing set contained about 90K functions.

More specifically, we retrieved our malware from repository *threatland/TL-BOTS*[2], which contains source files of a vast array of botnet families from 2014 to the present day. Our malware data set spans several malware families including Mirai, Gafgyt, Tsunami and Pilkah, which have been widespread in recent years [13, 135, 142, 143, 152]. Mirai, Gafgyt and Tsunami make up the majority, 89.74% of all ELF binaries that were submitted to VirusTotal between January 2015 and August 2018. [39]. Furthermore, security researchers have noticed new variants belonging to these malware families that are used to launch thousands of attacks in recent years [123, 124]. Source codes of over hundred variants of

[1] https://github.com/gsrishaila/DisCo-Combining-Disassemblers-for-Improved-Performance.git
[2] https://github.com/threatland/TL-BOTS

malware belonging to these families have been traced back to online repositories [39]. Hence, we believe that our dataset is representative of the malware found in the wild.

Since malware binaries may share certain characteristics that could be absent in benign software, we also show the generalizability of *DisCo* by applying it on a small set of benign binaries from SPEC 2017 benchmark. We discuss this in more detail in section 4.5.1. We also discuss the coverage of the data set in section 5.4.

As shown in figure 4.1, we show the effectiveness of combining various disassemblers effectively through our comprehensive study. Our study includes (a) various disassemblers, and (b) various compilation configurations and architectures.

4.2 Disassemblers and Compilation Configurations

The five baseline disassemblers: We consider 5 disassemblers, Angr [150], IDA Pro [66], Ghidra [108], BAP [23], and Radare2 [111] in our work. IDA Pro performed the best in previous studies and other disassemblers have been used in recent evaluation studies [72, 10]. We consider the following configurations and options.

a. Architectures: We consider two architectures, ARM version 5 and MIPS R3000. We focus on these architectures because the majority, 66.0% of the ELF malware binaries, belong to these architectures according to VirusTotal database [39]. Each architecture has different assembly language which requires different method and tools.

b. Compilers: We have compiled each program with two compilers: GCC version 5.5.0 and the Clang version 9.0.

For the remainder of the paper, we will use GCC to refer to GCC version 5.5.0, Clang to refer to Clang version 9.0, ARM to refer to ARM version 5 and MIPS to refer to MIPS R3000.

c. Five compilation optimization levels: For each architecture and for each compiler, we have considered 5 compiler optimization levels: O0, O1, O2, O3, and Os. Each level optimizes the binary in the three-way trade-off between compilation time, execution time, and the size of the binary.

d. Stripped and unstripped binaries: We only report results on stripped binaries, because the performance of some disassemblers deteriorates significantly when binaries are stripped [59]. Furthermore, around half of all ELF malware are stripped [39].

Finally, we focus on non-obfuscated binaries, as most disassemblers are not designed to work for obfuscated binaries. A recent large scale study shows that most IoT malware is non-obfuscated and unpacked [39]. We will apply *DisCo* on obfuscated malware in future studies, where we conjecture that combining disassemblers could be more effective due to the poorer performance of individual disassemblers.

4.3 Motivating Case Study

We present the key design and implementation ideas behind our approach. We start by presenting a case study that motivates and informs the design of our approach.

Motivating case-study: Disassemblers "see" different things. Combining the baseline diassemblers will only be beneficial if each disassembler recovers different parts of the binary structure. Individual disassemblers can miss some function starts (false negatives)

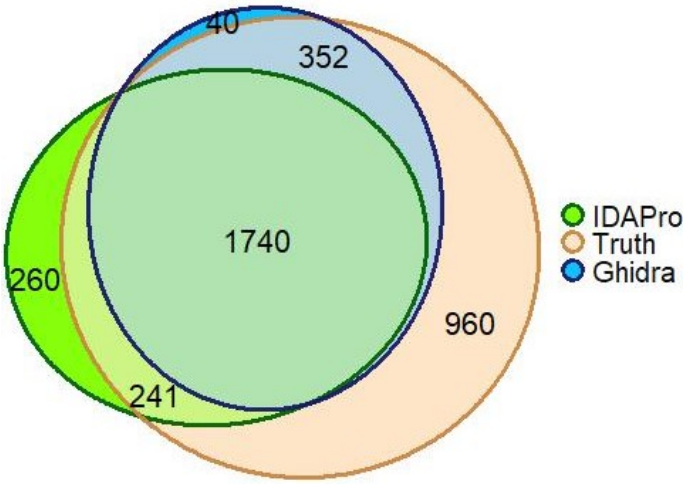


Figure 4.2: **Motivating observation:** Disassemblers complement each other. IDA Pro identifies 241 function starts and Ghidra 352 that the other does not identify. Similarly, the falsely identified function starts (260 and 40) seem to be disjoint. An efficient combination could improve the overall performance.

or erroneously identify a function start where there isn't one (false positive). Our intuition suggests that different disassemblers should have complimentary capabilities because they use different algorithms to identify the structure of the binary. We provide evidence that disassemblers produce different results, which enable the superior performance of *DisCo* as we see later in the paper.

We use IDA Pro and Ghidra to disassemble a random subset of binaries in our data set with focus on the *CFS* metric. IDA Pro has a 88.4% precision and and 60.2% recall. The corresponding values for Ghidra is 98.1% and 63.5%. Note that the numerical difference alone does not prove complementary capabilities, because the IDA Pro output may be subsumed by the Ghidra output, providing no additional information.

Observation 1. The two disassemblers have complementary results. Each disassembler correctly identified certain function starts that the other disassembler

missed. Specifically, 7.3% of the actual function starts were identified only by IDA Pro, while 10.7% of the function starts were identified only by Ghidra.

Observation 2. Combining the results should be done carefully. While this finding supports our intuition, it also shows that taking a simple union of the outputs from disassemblers will not necessarily guarantee optimal performance, especially for precision. This is because 11.6% and 1.9% of the functions starts found by IDA Pro and Ghidra were false positives.

We confirmed this observation by using two straightforward approaches. First, we applied a simple union of the outputs from disassemblers. We took a set of 58 MIPS binaries that were compiled with the Clang compiler at the O3 optimization level and applied the simple union technique on the outputs of all 5 disassemblers. This technique gave a precision of 48.6% and recall of 93%. This result shows that taking a simple union of all disassembler outputs will lead to low precision due to the presence of many false positives. Second, we considered the majority voting ensemble, in which a function start needs to be approved by a majority of the disassemblers. We repeated our experiment on the same set of binaries. This approach gave a perfect precision of 100%, but a recall of 65.6%. The low recall is because certain functions starts are only identified by a few of the disassemblers.

These initial results show the need for a more intelligent combination method, which we present below.

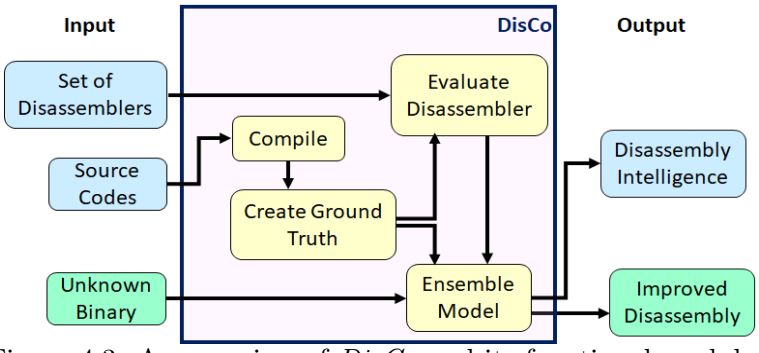


Figure 4.3: An overview of *DisCo* and its functional modules.

4.4 System Design and Implementation

We show the high-level architecture of *DisCo* in Figure 4.3. The blue boxes correspond to data used in the training phase, while the green ones are used in the testing phase. The yellow boxes are modules of *DisCo*.

a. Creating the Ground Truth: Before we evaluate and combine the disassemblers, we need to create training and testing datasets. To achieve this, we start with malware source code, but even then establishing the ground-truth requires some instrumentation and effort.

To obtain ground-truth, all the binaries are compiled with the -g option to attach richer debugging information to the resultant binaries. We use the DWARF library to identify function start addresses. We used a python script with imported DWARF libraries to create the ground-truth for each of our 1760 binaries.

b. Evaluating the disassemblers: Each binary in the training set is disassembeled with each disassembler. We compare the outputs from each disassembler with the ground-truth and use these results to: (a) create the training data for the ensemble model,

and (b) to extract useful information, which we can be used to improve each disassembler. This information is represented by the Disassembly Intelligence box in figure 4.3.

c. Creating the ensemble model: An ensemble model gives superior prediction performance by integrating multiple models effectively [120]. The goal of *DisCo*'s ensemble model is to combine the complementary disassembler capabilities in a way that increases the recall (identifies more real function starts) and precision (less misidentifications) compared to what is achieved by any of the disassemblers individually. A well-crafted ensemble model discovers complex correlation patterns in the output of the individual disassemblers and recognizes context specific strengths and weaknesses to combine these outputs effectively.

There are various ways to implement an ensemble model with respect to how the baseline disassemblers' output is combined. We decided to use stacking, where the output is combined using a neural networks [144]. A stacked ensemble model learns the best way to combine classification labels from multiple models. While other choices could be considered (e.g., majority vote), stacking achieves superior performance for many applications [3, 105], and works well in our context as well.

A realistic problem assumption. To emulate a realistic scenario, we assume that we only know the architecture for a given binary. In other words, we only know if a binary is compiled for MIPS or ARM. We do not assume knowledge of the compiler or compilation level used to produce the binary. As a result, we develop two analysis engines (and two ensemble models), one for each architecture.

The initial ensemble model. In the initial design, we used only one type of input, boolean values that represents whether a particular disassembler detected a particular func-

tion start. However, this model did not provide good results. Our investigation revealed that this information was insufficient and that additional context information from binary is required for effective predictions.

The context-aware ensemble model. We develop a "context aware" ensemble model by including information around the candidate function start location. For the MIPS architecture, we include two instructions before and after the potential function start. Since, each instruction is four bytes, we include 8 bytes before and after the potential function start. For the ARM architecture, we only include the 8 bytes after the candidate function start. The reason is that often there is inline data between functions in ARM binaries, and hence, 8 bytes before a function start may be data bytes, which adds noise to the process. Note that we treat the bytes (which correspond to binary instructions) as categorical data, since the numerical value of an instruction does not convey any additional information (other than the identity of the instruction).

d. Using *DisCo*: *DisCo* can be used by disassembly developers, malware analyst or security practitioners.

	Relative Perfomance Improvement (RPI)							
	Average				**5PWC**			
	MIPS		**ARM**		**MIPS**		**ARM**	
	GCC	**Clang**	**GCC**	**Clang**	**GCC**	**Clang**	**GCC**	**Clang**
DisCo(All)	**17.8**	**12.7**	**11.9**	**8.0**	**27.5**	**19.1**	**16.3**	**12.3**
DisCo(Free)	12.4	6.2	8.7	4.5	19.9	12.5	10.7	6.2
DisCo(IdaGhi)	5.8	2.5	7.2	5.2	8.3	7.9	12.5	9.1

Table 4.1: **The Relative Performance Improvement can be substantial:** We show the Relative Performance Improvement for ARM and MIPS for binaries compiled with GCC and Clang (-03 compilation level). The combined solution of *DisCo* is a significant improvement over the best contributing disassembler for both the average and 5PWC.

1. Accurate disassembly: A practitioner can use *DisCo* to analyze a malware binary of interest. She can use the platform, which we intend to open-source: she just needs to provide the binary to obtain the results as shown in figure 4.3. Alternatively, she can develop (or fork) her own version and expand with additional training data and disassemblers.

2. Improving other disassemblers: Developers can use *DisCo* to improve the performance of other disassemblers. First, we can provide the cases where a disassembler failed. This information alone can help the developers improve their approach. Second, *DisCo* can also generate new information that a disassembler can include in its knowledge base. For example, in the case of function start identification, *DisCo* can provide byte patterns that can be used as function start signatures. These signatures can be incorporated into the database of the disassembler. Ghidra has a well-defined interface for accepting such external information, which is why we selected it to showcase this capability, as we explain later in this section.

4.4.1 Implementation issues

We elaborate on some key implementation details for the choices made in building our instance of *DisCo*. These choices were done carefully and deliberately, and lead to good results demonstrating the promise of the approach. In the future, we will consider more options and different ways to fine-tune the performance further as we discuss in section 5.4.

Creating the training and test sets: The training set for each model is created by allowing all disassemblers in the group to identify functions from 600 binaries compiled from the 30 malware programs in our training set. The testing set consists of 1160 binaries

compiled from 58 other malware programs in our testing set. These binaries were compiled with 2 compilers, GCC and Clang and with five compilation levels, O0,O1,O2,O3,Os. Note that we create and train two different models for each of our two architectures.

We disassemble each binary by using each of the five disassemblers to get the training and testing input for our model. We create a set containing functions start locations that were identified by at least one disassembler. We use a boolean value to represent the "vote" of each disassemblers for each candidate function start. We also record 8 bytes after each function start. For MIPS binaries, we also record 8 bytes before each function start as explained before. We provide these inputs into the model.

Deploying the disassemblers. Some disassemblers can be used with different operational options, and we selected the most optimal options based on previous work [72]. Furthermore, in the case of Ghidra v9.1 we discovered a bug, which affected its performance for the ARM architecture. We discuss this in the next section. Interestingly, that bug was not present in version v9.0.4. We opted to give Ghidra the "benefit of the doubt" and used version v9.0.4 for the ARM architectures, and version v9.1 for the MIPS architecture.

Note that Angr fails to complete disassembly for some binaries, and terminates without output, as has been observed in previous studies [72]. We use and report only the cases where Angr disassembles a binary successfully.

Combining different disassemblers: three *DisCo* variants: Our approach can combine any number of disassemblers that a practitioner would have available, as long as they can be included in our training pipeline.

We consider three variants of *DisCo* as our focus is to show the potential of harnessing the collective capabilities of different groups of disassemblers.

***DisCo(All)*:** We considered all five disassemblers: Angr, IDA Pro, Ghidra, BAP, and Radare2.

***DisCo(Free)*:** We considered only the non-commercial disassemblers: Angr, Ghidra, BAP, and Radare2.

***DisCo(IdaGhi)*:** We considered two disassemblers: IDA Pro and Ghidra.

We share more details about the models used for *DisCo(All)*. The ARM model was trained on 26K functions and tested on 50K functions. The MIPS model was trained on 28K functions and tested on 40K functions.The function starts in the training and testing set for each architecture differ because the number of function starts missed by all disassemblers and the number of falsely identified function starts identified by each disassembler for each configuration varies. We used a feedforward based neural network with 2 hidden layers for each model. The first layer had 1000 nodes while the second layer had 250 nodes. We used these parameters because they gave the most optimal results when we used the 10 fold cross validation to train the model.

We created the models for *DisCo(Free)* and *DisCo(IdaGhi)* in a similar way. The number of functions used to train and test the model will be lesser than the number used for *DisCo(All)* because we are combining lesser number of disassemblers. Hence, the training time required was also lesser.

The output of these *DisCo* versions produces the results shown as *Improved Disassembly* in figure 4.3. Note that it is well established that Radare2 does not support ARM binaries compiled with Clang [72].

Improving disassemblers: the case of *Ghidra+*. As a proof of concept, we show how *DisCo* can improve disassemblers focusing on Ghidra. *DisCo* can generate function starts signature by using the training data which we include in Ghidra's database.

We use two *DisCo* variants, *DisCo(Free)* and *DisCo(All)* to improve Ghidra 9.1. The improved versions of Ghidra are called *Ghidra+(Free)* and *Ghidra+(All)* respectively. As we will see later, both improved versions perform overall significantly better compared to the original Ghidra.

4.5 Experimental Evaluation

We evaluate the effectiveness of our approach with the datasets which we described in section 4.1 using the ground-truth which we discussed in section 4.4.1. We group our experimental results around the following three questions.

Q1: How beneficial is the combination of disassemblers?

Q2: How can our approach improve a disassembler?

Q3: Which factors affect disassembler performance?

We answer each question with a series of observations. Going one step further, we also provide a set preliminary investigations into issues that include the performance of our approach on benign malware.

	MIPS		ARM	
	GCC	**Clang**	**GCC**	**Clang**
DisCo(All)	**99.8**	**99.5**	**98.9**	**99.3**
DisCo(Free)	94.4	93.0	95.7	95.8
DisCo(IdaGhi)	82.6	87.8	94.2	96.5
Ghidra+(All)	90.4	88.6	91.0	98.1
Ghidra+(Free)	90.1	87.4	91.7	98.0
Angr	**82.0**	**86.8**	50.2	43.9
BAP	56.9	62.1	63.6	47.0
Ghidra	76.8	85.3	**87.0**	**91.3**
IDA Pro	71.8	73.0	79.1	76.0
Radare2	68.5	65.6	74.0	NS

Table 4.2: ***DisCo* combines disassemblers effectively:** We show the average *CFS* F1 score for binaries compiled with the O3 optimization level. *Ghidra+* shows significant improvement over Ghidra. NS means not supported.

Q1: How beneficial is the combination of disassemblers? Our results suggest that combining the disassemblers provides significantly superior performance. We provide the RPI values for each *DisCo* variant in table 4.1. In tables, 4.2 and 4.3. we show the performance of *DisCo* variants, *Ghidra+* variants and other disassemblers for *CFS* metric. Table 4.3 shows both average and 5PWC performance for the MIPS binaries. All tables show the results for binaries compiled with the O3 compiler optimization levels. Despite having extensive tables for all configurations, we are not able to show them due to space limitations. As we already discussed in the previous section, Radare2 does not support ARM binaries compiled with Clang. Hence, it does not make sense to include in the combined solution.

Observation 1: Combining disassemblers provides significant improvement for both average and worst case. Combining the collective wisdom of disassemblers leads to significant improvements, often in the double-digits, in both the average and the worst case metrics in our experiments. In table 4.1, we see that each of our *DisCo*

variants give consistent positive RPI values of up to 17.8% for the average F1 score and up to 27.5% for the 5PWC F1 scores for all configuration options. This shows that we have combined the disassemblers efficiently.

First, we see a significant performance increase on the average F1 score of function start identification. In table 4.2, we see that *DisCo(All)* gives an average F1 score of 98.9% or above for binaries of both architectures and compilers. When compared with the best performing disassembler, this is an improvement of up to 17.8% for GCC and 12.7% for Clang binaries in the MIPS architecture. The corresponding values for ARM binaries are 11.9% for GCC and 8.0% for Clang.

Second, we see a significant improvement in the worst case performance as this is captured by the 5PWC metric of the worst performing binaries for each disassembler. In table 4.3, we see that *DisCo(All)* provides an improvement of 27.5% for GCC and 19.1% for Clang for the MIPS binaries compared to the 5PWC of our individual disassemblers. We also see that the other *DisCo* variants also provide significant improvements, though smaller than that of *DisCo(All)*.

Observation 2: Each disassembler seems to add value to the union. Unsurprisingly, combining more disassemblers leads to better average and 5PWC scores. In table 4.2, we see that the performance improves when we go: (a) from *DisCo(IdaGhi)* to *DisCo(All)*, and (b) from *DisCo(Free)* to *DisCo(All)* for both architectures and different compilers. Note that comparing *DisCo(Free)* and *DisCo(IdaGhi)* is less straightforward, as the Free group does not include IDA Pro. In table 4.3, we see even larger improvement for the 5PWC values between *DisCo(All)* and the other two *DisCo* variants.

	GCC		Clang	
	Aver.	**5PWC**	**Aver.**	**5PWC**
DisCo(All)	**99.8**	**99.0**	**99.5**	**96.4**
DisCo(Free)	94.4	91.4	93.0	89.8
DisCo(IdaGhi)	82.6	73.8	87.8	78.0
Ghidra+(All)	90.4	75.0	88.6	78.5
Ghidra+(Free)	90.1	73.0	87.4	78.5
Angr	**82.0**	**71.5**	**86.8**	**77.3**
BAP	56.9	15.3	62.1	28.1
Ghidra	76.8	57.2	85.3	70.1
IDA Pro	71.8	65.5	73.0	63.5
Radare2	68.5	45.6	65.6	45.0

Table 4.3: **Combining the disassemblers improves the worst case performance significantly:** We show the average and 5PWC F1 score for function starts, *CFS*, for binaries compiled with the O3 optimization level for MIPS. *Ghidra+* also shows significant improvement in its worst case performance compared to Ghidra.

The more interesting observation is that even the disassemblers that do not perform well on their own seem to still add value when included in the ensemble model. The improvement is more pronounced for the worst case performance metric. An indication of this can be found in table 4.3. If we rank the disassembler performance for MIPS binaries based on the average scores in decreasing order, we get Angr, Ghidra, IDA Pro, Radare2 and Bap. Although IDA Pro is the third best performing disassembler, including it in the combination, namely going from *DisCo(Free)* to *DisCo(All)*, improves the average F1 score and 5PWC by as much as 6.5% and 6.6% for Clang respectively.

We observe a similar phenomenon in table 4.2. Ghidra and IDA Pro are the best performing disassemblers for binaries of the ARM architecture. However, the performance of *DisCo(All)* outperforms *DisCo(IdaGhi)* by up to 4.7% for this architecture. In other words, even adding the three lower-performing disassemblers leads to improved performance.

Observation 3: Disassembler performance varies significantly across binaries. As mentioned in the previous chapter, we noticed that disassembler performance varies greatly even for binaries of the same configuration. In table 4.3, we observe that the differences between the average and the 5PWC F1 scores range widely between 6.3-41.5% for GCC and 9.5-34% for Clang for MIPS binaries. This suggests that the disassembly accuracy that we can expect from a disassembler can vary greatly across binaries.

More interestingly, we observe that the "relative ranking" of the disassemblers can vary per binary. In fact, it is not unlikely that a lower-ranked disassembler based on average performs better for a number of binaries compared to a higher-ranked disassembler. For example, in table 4.3, we see that Ghidra outperforms IDA Pro by 5.0% in terms of the average F1 score for MIPS binaries with GCC. However, IDA Pro outperforms Ghidra by 8.3% for the 5PWC score. Intrigued by this, we looked at individual binaries. We found that IDA Pro outperformed Ghidra in 25.9% of the binaries in this dataset and by at least 10%. Therefore, answering the question "which is the better disassembler for a specific binary?" does not have an easy answer, even if we know the average performance.

Q2: How can our approach improve a disassembler?We show how the extracted wisdom of a group of disassemblers can improve an individual disassemblers.

Observation 4: Information from *DisCo* improves Ghidra substantially. As mentioned in section 5.2, we use the *DisCo* model to update the function signature byte pattern file for any disassembler which can utilize such information. We use *DisCo(Free)*

and *DisCo(All)* to create two new versions of Ghidra, *Ghidra+(Free)* and *Ghidra+(All)* respectively.

Our results show that *Ghidra+(All)* exhibits considerable improvement over Ghidra for both the average and 5PWC scores. In table 4.2, we see that *Ghidra+(All)* improves the performance of Ghidra by up to 13.6% for the MIPS binaries and by up to 6.8% for the ARM binaries. In table 4.3, we see that *Ghidra+(All)* also improves the 5PWC scores by 17.8% for the MIPS binaries compiled with GCC and by 8.4% for MIPS binaries compiled with Clang. *DisCo(All)* added an additional of 1696 signatures for the ARM architecture and 3418 signatures to the MIPS architecture in *Ghidra+(All)*. Overall, *Ghidra+(Free)* also exhibits similar improvements over Ghidra with two *Ghidra+* versions differing by a maximum of 2% for the average and 5PWC scores.

We wanted to compare the performance between an improved Ghidra and the *DisCo* that helped it improve by providing signatures. It turns out that the *DisCo* variant typically performs better in most cases, but not all! Both *DisCo* variants perform better than the corresponding *Ghidra+* variants in all the cases except for *DisCo(Free)* for the ARM binaries compiled with Clang. We show the results for the O3 compilation level in tables 4.2 and 4.3, while qualitatively similar results were obtained for other compilation level. *DisCo(All)* outperforms *Ghidra+(All)* by 1.2 - 10.9% while *DisCo(Free)* outperforms *Ghidra+(Free)* by 4.0 - 5.6%. Interestingly, in the case of ARM Clang, *Ghidra+(Free)* performs better than *DisCo(Free)* by 2.2%, but note they both perform better than the original Ghidra. The usual superior performance of the *DisCo* variant over *Ghidra+* variant is not surprising. The neural network model can find complex relationships between the

inputs and the outputs. All these complex relationships of the ensemble model cannot be fully captured by the byte patterns of function signatures. We will investigate the case of ARM Clang in the future, especially since our other investigations often led to interesting insights.

Q3: Which factors affect disassemble performance? As mentioned in the previous chapter, **disassembler performance is affected significantly by all three factors: (a) architecture, (b) compiler, and (c) compilation levels.** Our results in table 4.2 and 4.3 reveals several insights.

a. The effect of the architecture: Some disassemblers have better support for binaries belonging to one architecture compared to another. The more striking case is Angr, which gives an average F1 score in the 80s for MIPS binaries, in contrast to 40s and 50s for ARM binaries and this applies to both GCC and Clang as shown in table 4.2. The effect of the architecture is interesting for BAP as it depends on the compiler: BAP with Clang does better than GCC in MIPS, but BAP with Clang does worse than GCC in ARM. The exact values are shown in table 4.2.

In general, a practitioner need to be mindful of this kind of variations for different architectures.

b. The effect of the compiler: The compiler affects the performance for some disassemblers significantly for both the average and the worst case. For example, BAP has an average F1 score of 63.6% for ARM binaries with GCC and only 47.0% for ARM binaries with Clang (see table 4.2).

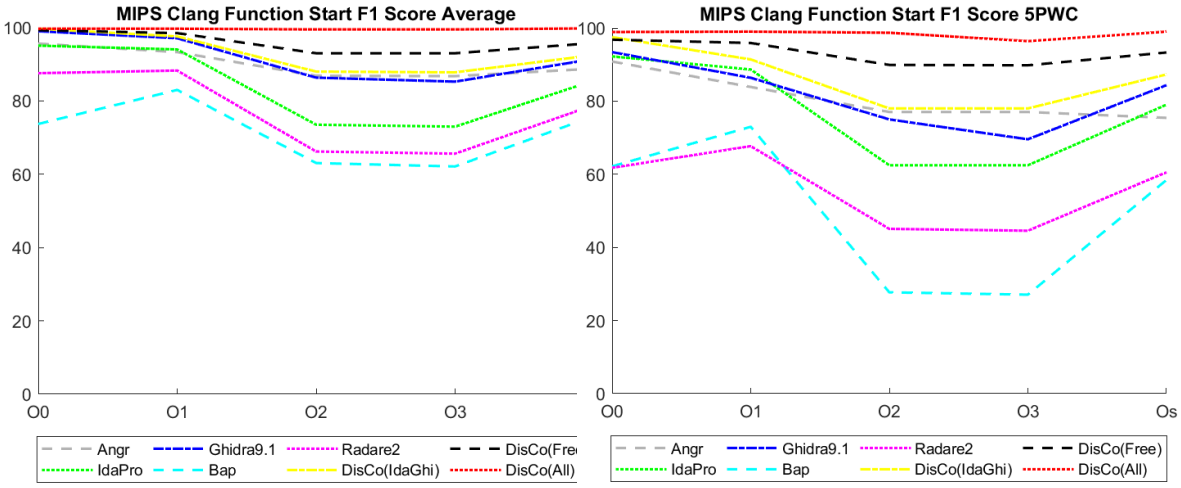


Figure 4.4: Average and 5PWC for F1 Score for Function Start Identification for MIPS binaries compiled with Clang.

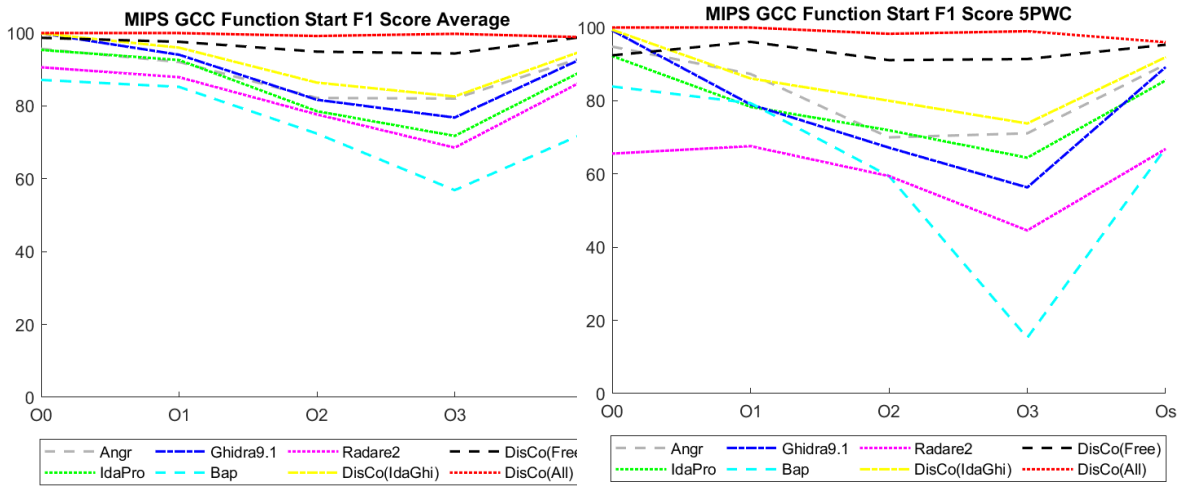


Figure 4.5: Average and 5PWC for F1 Score for Function Start Identification for MIPS binaries compiled with GCC.

Similarly, we see that Ghidra performs better for Clang for both MIPS and ARM. For MIPS, its performance increases from 76.8% for GCC to 85.3% for Clang on average, and from 57.2% for GCC to 70.1% for Clang in the worst case.

c. The effect of the compiler optimization level: Disassembler performance is affected by the compiler optimization levels used during compilation significantly. Most disassemblers tend to perform worse when binaries are compiled with the O2 or the O3

compilation level. Figures 4.4 and 4.5 illustrate this for average and 5PWC F1 score for MIPS binaries compiled with Clang and GCC. Similar trends have also been observed for ARM binaries. Figures 4.6 and 4.7 illustrate this for average and 5PWC F1 score for ARM binaries compiled with Clang and GCC.

Our results show that *DisCo(All)* is less sensitive to compiler optimization levels compared to other disassemblers. Even when we consider the best disassembler for each architecture, the difference in mean F1 scores for the various optimizations is 10.1% for the

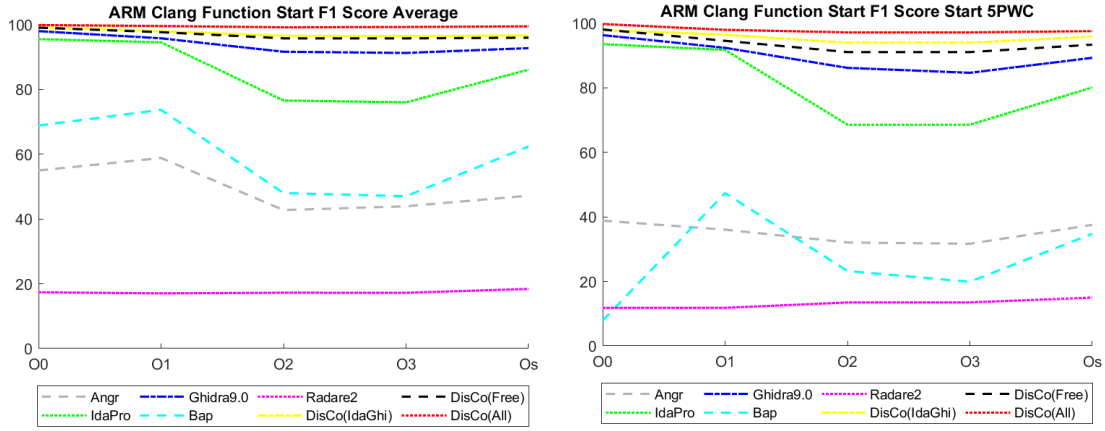


Figure 4.6: Average and 5PWC for F1 Score for Function Start Identification for ARM binaries compiled with Clang.

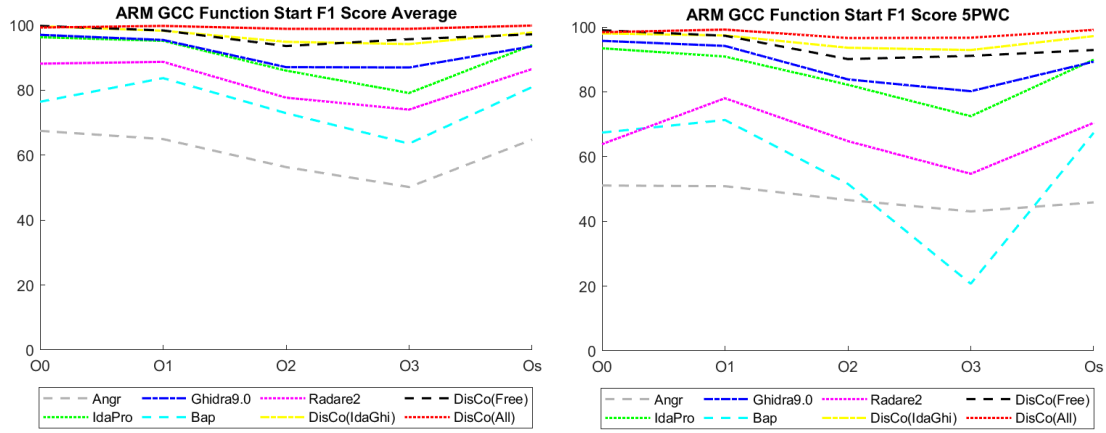


Figure 4.7: Average and 5PWC for F1 Score for Function Start Identification for ARM binaries compiled with GCC.

ARM architecture and 13.7% for the MIPS architecture. In contrast, *DisCo(All)* reduces this difference to 1.0% for the ARM binaries and 1.2% for the MIPS binaries. This increase in the reliability of the results is yet another argument in favor of the value of combining disassemblers.

The overarching conclusion further supports the benefit of *DisCo*: the performance of disassemblers is affected by configuration options. As a result, it is hard to pinpoint a single "best" disassembler that will perform well in all configuration scenarios. In addition, we see that *DisCo* is minimally sensitive to these variants: as the 5PWC is very close to the high average performance. In other words, *DisCo* offers good performance reliably with small variation across many different configurations.

4.5.1 Some Exploratory Investigations

We further evaluate *DisCo* by testing its capabilities in the following situations: (a) limited training data, and (b) benign binaries. Note that due to space limitations, this is mostly a preliminary study, which we intend to substantiate in future.

We show that *DisCo* performs well in these situations by using *DisCo(IdaGhi)*. In this subsection, we focus more on *DisCo(IdaGhi)* as an instantiation of *DisCo*. We create the training set by compiling 16 malware source codes. *DisCo* was created by combining the outputs from IDA Pro and Ghidra from the training set binaries. Figure 4.8 shows the results when we evaluated *DisCo* on 4 malware binaries compiled from 4 other malware source codes and 9 benign binaries from the SPEC2017 benchmark. All binaries were compiled with GCC from C language source codes with O3 option for the MIPS architecture.

Preliminary Investigation 1: How sensitive is *DisCo* to the training set size? We conduct the following experiment to assess the sensitivity of *DisCo* to the training set. We use a subset of 16 malware source codes for training. In this case, we focused on *DisCo(IdaGhi)*, since combining only two disassemblers could stress test the capabilities of *DisCo* using the MIPS GCC scenario. It turns out that even in this case the performance was able to improve the F1 score by 6.7% for the malware binaries which is comparable to the 5.8% improvement we saw in our larger training dataset. This initial experiment suggests that *DisCo* can perform well with limited number of training data. We intend to study how performance is affected by the size of the training data in future.

Preliminary Investigation 2: Does *DisCo* work well for benign binaries too? We wanted to see if our approach can work well for benign binaries. *DisCo* was able to improve the F1 score by 4.4% for the benign binaries respectively. This suggests that the performance improvement by *DisCo* could also apply to benign binaries. Note that here *DisCo* was trained on malware binaries. We will further investigate if by training on benign binaries would bring the performance improvement closer to the improvement we saw with malware binaries.

Preliminary Investigation 3: Are malware binaries harder to disassemble than benign binaries? Evaluation results based on the limited set of malware and benign binaries suggest that both disassemblers, IDA Pro and Ghidra perform better in F1 score for the *CFS* metric for benign binaries. IDA Pro performs better for benign binaries by 7.7%, while Ghidra performs better for benign binaries by 7.4%. A possible reason for this could be using benign binaries from well known benchmarks to test disassemblers.

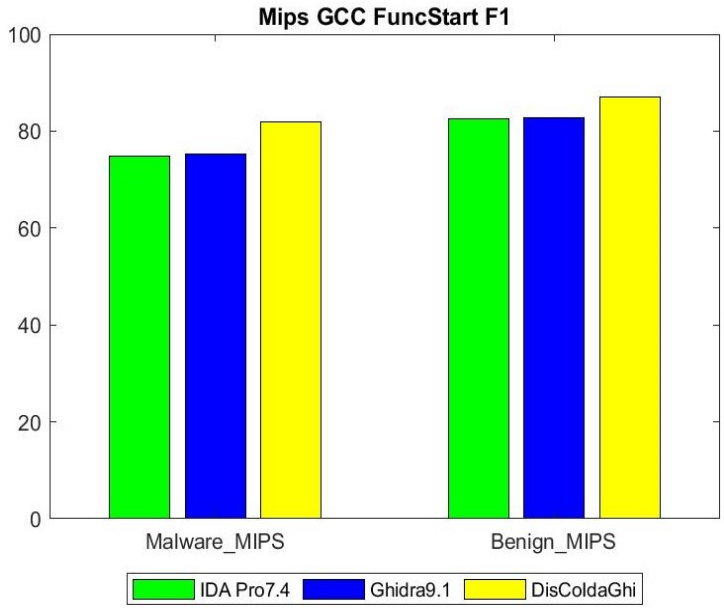


Figure 4.8: **Benign and malware binaries:** *DisCo* improves the performance even in the case of benign binaries. *DisCo* improves the F1 score by 6.7% for malware and 4.4% for benign binaries. Disassembling benign binaries seems easier. The reported results are for the MIPS architecture with GCC and the O3 compilation level.

4.6 Discussions And Future Work

In this section, we discuss the broader context and limitations of our work.

How will *DisCo* be used in practice? As we already mentioned, *DisCo* provides: (a) more accurate disassembly for a given binary, and (b) information to improve individual disassemblers. Therefore, we envision two different types of users: (a) security practitioners, who want to understand a malware binary, and (b) developers of disassemblers. Users can either use our approach to instantiate their own version of *DisCo* or use our own open-source version of the tool. Note that using commercial disassemblers will require a license. Developers of disassemblers can use *DisCo* as a mechanism to evaluate their tool, compare their tool with other tools, and extract information that can improve their tool. We saw a case study of this in the previous section where we improved Ghidra.

Furthermore, we enthusiastically invite the community to help improve and extend our approach by: (a) introducing more capabilities, such as adding disassemblers, and (b) adding more samples to the training and testing datasets.

Can we improve the performance of *DisCo* further? Although the initial results are significant, there are ways to farther improve the performance of our approach. In certain cases, *DisCo* misses identifying function starts. The operands of some instructions around the function start can be one of multiple registers. We conjecture that if the training set does not contain all variants of these instructions, with the various possible registers as operands, the model can miss recognizing these kinds of function start byte patterns. Increasing the size of training data may help to address this problem. Finally, we can also improve performance by including more disassemblers.

Our ultimate goal is to include as many disassemblers as possible in *DisCo*, which will strengthen the combined performance. In future, we plan to add two more disassemblers, Hopper [48] and Binary Ninja [70], to further improve the combined performance. We can share our preliminary results with Binary Ninja [70]. We report the results from the MIPS architecture here for the O3 compilation level. Binary Ninja performed very well in our dataset with F1 scores of 92.6% for GCC and 95.7% Clang for MIPS. Combining all six disassemblers leads to an RPI 4.7% for GCC and 3.5% Clang on the average performance for MIPS. In addition, the 5PWC of the combined performance showed a more significant improvement: 10.8% for GCC and 11.9% Clang. Recall that previous work found that IDA Pro performed better than Binary Ninja in their dataset [72]. These variations further highlight the benefit of combining disassemblers.

Is there an "optimal" set of disassemblers to combine? Our position is to sidestep the *best-disassembler* question and instead use the collective power of all the disassemblers that one can afford to purchase and integrate in a *DisCo*-like approach. The motivation is that each disassembler can provide useful and unique information for various compilation configurations. However, it is natural to ask for the minimum set of disassemblers to provide great performance. To answer, an extensive study that encompasses: (a) source code variations, in terms of programming approaches, styles and application types, and (b) various compiler configurations is needed.

Can *DisCo* improve each disassembler? We argue that *DisCo* is a systematic approach to evaluate and cross-pollinate disassemblers to improve each one. First, we showcased this capability when we improved Ghidra by adding function start signatures in its knowledge-base. Second, our systematic approach can pin-point systemic weaknesses in the disassemblers. As we saw, we found a bug in Ghidra v9.1. We envision this knowledge-transfer and evaluation operation to be infrequent. For example, it can take place every three to six months, or be prompted by events, such as new releases of the disassemblers, or the addition of new training data in *DisCo*.

We chose to use 8 bytes around function starts in *DisCo* as input to the model for the following reasons. In the ARM and MIPS architecture, each instruction is 4 bytes long. We started by looking at byte patterns used in Ghidra. We noticed that for the ARM architecture, most rules tend to use 8 to 16 bytes around the function start with varying numbers of instructions taken before and after the function start. For the MIPS architecture, most rules use 8 to 20 bytes before the function start and 8 to 12 bytes after

the function start. Our goal was to to use the minimum number of bytes, which can shorten the training time and reduce the possibility of overfitting. Hence, we decided to start with 8 bytes before and after the function start. Since we obtained good results by using 8 bytes, we did not explore using other numbers of bytes.

How much can our approach generalize? Our goal here is to introduce the idea of combining disassemblers as a new way of thinking about disassembling and show that it leads to promising results. To obtain these results, we had to focus on specific choices of compiler configurations such as MIPS and ARM architectures, C language programs, and focused on the *CFS* metric. A natural question is how much can we generalize this approach. We are confident that our approach can extend and generalize to: (a) any number of disassemblers, (b) binaries of various architectures, (c) different compilers and compilation options, and (d) different programming languages. One possible extension of our work is to apply our technique to a variable length Instruction Set Architecture, (ISA) like x86. In such a scenario, we could decide on the number of bytes used as inputs to the model by referring to open sourced disassemblers or by experimenting with various numbers of bytes. Each of these extensions vary in the required effort.

Are our datasets representative? This is the typical and fair, hard question for any evaluation study. First, we made a point to include in our dataset a number of the most prominent and recent malware families, as we saw in section 4.1. Second, our goal is to show the ability of our approach to leverage the merits of each disassembler. We argue that the "intelligence" of our algorithm is second-order question: a different dataset may affect the individual performance of each disassembler, but that does not affect the

	Avg. time for MIPS binaries(s)	Avg. time for ARM binaries(s)
Angr	26.7	12.0
BAP	5.7	5.1
Ghidra	32.1	28.2
Ghidra+(All)	31.2	30.6
IDA Pro	29.4	9.9
Radare2	1.2	1.2

Table 4.4: **Time requirements for various disassemblers:** We show the average time required for each disassembler for each binary.

capability of *DisCo* to combine these performances. Of course, if the algorithms perform badly, the combined performance will be lower than what we saw here. In other words, the disassemblers need to keep up with the malware intricacies, as *DisCo* is simply leveraging their combined capability.

For what types of binaries does *DisCo* work well? Our work focuses on IoT malware binaries. This influences our choices in terms of architectures, compiler, and training and testing datasets. While there are various types of binaries and platforms we can consider, the overarching statement is that combining different disassemblers can only provide better results, if it is done efficiently with sufficient training.

Benign binaries. Disassembly of benign binaries can also benefit from a combined approach. First, we showed some promising initial results in section 4.5.1, where we tested on a small set of benign binaries from the SPEC 2017 without training for benign binaries. We plan to conduct a large scale study of *DisCo* on benign binaries.

Obfuscated binaries. Developers often obfuscate their binaries to impede one's ability to reverse engineer them. Note that most disassembler methods and related studies focus on unobfuscated binaries [10, 19, 9]. One recent works that considered some obfuscated

binaries found that obfuscation poses a challenge to disassembly tools and that different tools offer varying performance for these binaries [72]. Here, we did not consider obfuscated binaries on disassembly accuracy, but we intend to study this in the future.

What is the time requirement to use DisCo? If we want to use *DisCo* on a test binary then we need to extract some information from the binary from all five disassemblers in the group. The most time efficient way will be to use the disassemblers in parallel. We recorded the time taken for various disassemblers to analyze 400 binaries. Table 4.4 shows the average time needed by the various disassembler tools to analyze a binary. This time is the total time required by the disassembler to disassemble the binary and run python scripts to extract information from the binary that will be used by the DisCo model later. Hence, on average, when we use *DisCo(All)* for a given binary using our model, we can obtain outputs of all 5 disassemblers in 30.2 seconds if we operate them in parallel. The time taken to train the model for each architecture for *DisCo(All)* is 1 hour. *Ghidra+(All)*, on average requires 30.9 seconds to analyze a binary.

4.7 Related Work

To the best of our knowledge, there has not been any previous study that has combined the capabilities of disassemblers to improve disassembly accuracy. Furthermore, there is relatively limited prior work at the intersection of disassembling (a) *malware* binaries, and (b) the MIPS and ARM architectures. Since we have used an ensemble model to combine disassemblers, we also include a brief overview of studies on ensemble learning.We group other previous work into the categories below.

a. Evaluating Disassemblers: The more recent study [72] evaluates various disassemblers by using *benign* ARM binaries. They observe that various disassemblers offer different levels of accuracy for different types of programs. Another related work [10] evaluates the performance of disassemblers by using benign binaries for the `x86` architecture. Both works endorse IDA Pro as the best disassembler. In terms of malware binaries, a recent work [59] focuses exclusively on IDA Pro (version 6.8) and on a limited set of malware binaries. They found that malware authors tend to prefer to use the `-O{3}` options and that IDA Pro performs poorly for *CFS* for stripped binaries compiled with that option. That effort differs from our work significantly as: (a) it does not propose to combine disassemblers, and (b) it evaluates only IDA Pro in contrast to the five disassemblers that we use here.

b. Developing Novel Disassembly Techniques: Several studies propose efficient disassembly techniques, but we have not found any effort that attempts to combine multiple disassemblers.

A recent study [9] uses the control flow graph to improve function identification in stripped binaries. However, this technique tends to fail to identify functions called by using tail calls and can only be used in architectures with specific opcode for function calls, unlike ARM. Other works focus on other aspects of disassembly like security, speed and handling obfuscation in x86 binaries [160, 82, 85]. Other works present techniques like superset disassembly, probabilistic disassembly, and static analysis based method for x86 binaries [17, 97, 117]. Some works propose machine learning techniques for disassemebly [74] and to identify function starts [122, 16, 23, 132]. However, later works found that some of these works suffer from evaluation bias [9]. Other approaches use heuristics and or well-known

function signatures to identify function starts [83, 150, 133]. Another work [55] proposes a technique to translate assembly code into Intermediate Representation(IR) to recover control flow graphs and identify function boundaries for various architectures. Another work [113] combines probabilistic fingerprint of binary code with a probabilistic graphical model to match function names to program structure in stripped x86_64 binaries. A very recent work [96] introduces a technique to calculate the probability that an instruction would start at a certain address. Such techniques aim to improve the instruction recovery rates in architectures like x86 where instructions can have varied sizes. In contrast, assembly instructions found in the `ARM` and `MIPS` binaries have a fixed size of 4 bytes, so the start of the next instruction can be predicted. Another work [19], presents a speculative disassembly technique for THUMB binaries.

Commercial Tools and Platforms: There are many existing disassemblers that can analyze binaries of various architectures. Some examples include IDA Pro, Hopper, Dynist, BAP, ByteWeight, Jakstab, Angr, Ghidra and Binary Ninja. [66, 48, 92, 92, 23, 16, 78, 150, 108, 70].

Dynamic analysis and sandboxes: There are many efforts that use dynamic execution to analyze a malware binaries, which is a complementary approach to the static analysis, which is our focus here. Indicatively, we can mention a few recent efforts [44, 45, **?**, **?**] that create platforms that manage to activate IoT malware malware. Another work [28] develops an IoT sandbox which can support 9 kinds of CPU architectures including ARM and MIPS.

c. Previous studies on ensemble learning: A recent survey [125] reviews both traditional and newer ensemble learning techniques and analyzes the trends and limitations of these methods. Other works propose methods to quantify the benefit of using an ensemble for a set of classifiers [22, 81]. Another work finds the reason behind trends in test errors in voting methods [128].

4.8 Conclusion

The overarching novelty of the work is the idea of harnessing the collective power of the many disassemblers that are available in the security community. To substantiate this idea, we develop *DisCo*, a systematic approach to analyze and synthesize disassemblers. The goal is to achieve the best possible disassembling performance for IoT malware binaries. Hence, we focus on the ARM and MIPS architectures.

First, we show that *DisCo* can combine the collective power of disassemblers effectively as it consistently outperforms each individual disassembler. For example, our approach outperforms the best contributing disassembler by as much as 17.8% for F1 score for function start identification for MIPS binaries compiled with GCC with O3 option.

We then show that the collective power of the disassemblers can be brought back to improve each disassembler. We showcase this capability by developing *Ghidra+*, which outperforms the initial Ghidra by as much as 13.6% in terms of F1 score by simply using function signatures identified in our approach. In addition, our systematic evaluation within our approach led to a bug discovery: a bug introduced in Ghidra 9.1, for which the Ghidra team expressed appreciation.

Finally, we conduct a study to understand the effect that configuration and scenarios have on disassembly performance. We study the effect of the architecture, the compiler, and the compiler options affect the performance of disassemblers significantly. We find that the performance varies significantly, and find further evidence that there is no one single best disassembler especially if we consider performance per binary, and not just on average. This further supports the idea that combining disassemblers promises to provide significant advantage over each individual method.

The contribution of our work is three fold. Firstly, we present an evaluate *DisCo*, an ensemble of the five popularly used disassemblers for various compilation scenarios to combine their complementary capabilities. Our approach achieves higher performance than any individual disassembler. Secondly, we show how our model can be used to improve other disasssemblers and lastly we have shown that compiler options affect function start identification for most disassemblers.

Our work in perspective. Our work is a significant step in assessing existing and developing new capabilities in disassembling binaries, especially in the space of IoT malware. We hope to enable developers and users of such tools to make informed decisions leveraging both our system and the datasets that we have and will continue to develop. We plan to open-source and share all our tools and data and hope that this encourages further research in this direction.

Chapter 5

MalEvasion: Simple string manipulations derail malware detection

IoT malware malware has emerged as a serious threat. Over 41 billion IoT devices will be present worldwide by 2025 [25]. The Mirai attack in 2016 shows why we need to safeguard IoT devices. Mirai was first observed in August 2016. By October 2016, Mirai was able to successfully launch a DDOS attack against high-profile targets like Krebs on Security and Dyn [157, 14] with the help of a botnet. This resulted in major disruptions in Internet services across Europe and US that lasted for 83 hours. This attack left 131 000 IoT devices infected and resulted in a loss of $440K [109, 149].

Taking a hacker-centric view, we ask the following key question: "what are simple techniques that can be used to avoid being detected by VirusTotal engines?". This is the

question that motivates our work. In more detail, we frame the question with the following constraints and assumptions. First, we require evasion techniques to have the following characteristics: (a) be easy to apply and (b) preserve original functionality of the malware program. Second,since we assume the position of hackers, we have access to the malware source code. Third, we focus on the MIPS and ARM architectures because most devices affected by IoT malware belong to these architectures [6].

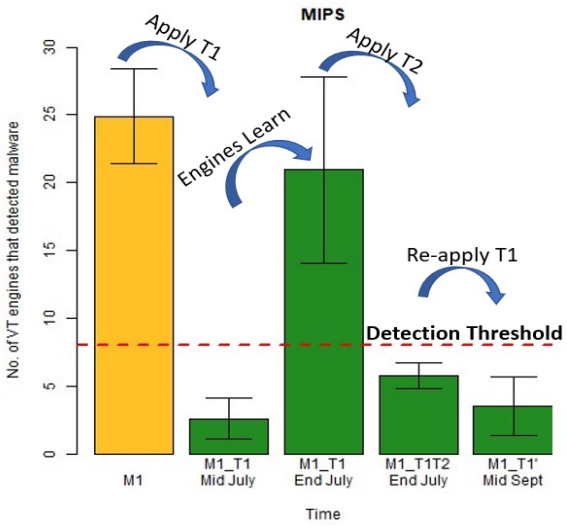


Figure 5.1: ***MalEvasion*: simple techniques can evade VirusTotal engines repeatedly and effectively on MIPS:** (a) applying technique T1 reduces the detection from roughly 25 to 3 engines, (b) the engines "learn" to recognize the submitted binaries going back to 21 engines within two weeks, (c) applying technique T2 reduces detection to 6 engines, (d) re-applying technique T1 (with different input parameter) works again reducing detection to roughly 4 engines. The red line is a commonly used threshold for arriving at a final determination for malicious nature of the binary.

An efficient way to evaluate the vast majority of anti-malware engines is through the widely-used VirusTotal platform [155]. VirusTotal is a free online service that analyzes binaries by using 71 anti-malware engines [148]. VirusTotal reports how each engine classifies a binary: (a) malicious or benign, and (b) malware family [147]. VirusTotal has been

extensively used in research to label datasets and evaluate tools [29, 58, 62, 75, 21, 139, 141, 163]. We are grateful to the VirusTotal team as their work makes our research study possible.

There is relatively limited work on the problem as we frame it here: unlike most previous work, we focus on: (a) modifying the source code, and (b) IoT malware. In more detail, we can group previous efforts in the following categories. One category focuses on manipulating the *binaries* to evade detection [89, 54, 7]. Another category focuses on evaluating the detection capabilities of VirusTotal engines [99, 163] by using a set of binaries, but without developing evasion techniques. We discuss related work in detail in section 5.5.

In this work, we make two main contributions. First, we develop *MalEvasion*, a framework that includes a set of simple but effective evasion techniques to evade anti-malware engines. Second, we conduct a systematic study of the robustness (or lack thereof) of 71 engines in VirusTotal using our framework and 1740 binaries. We study two functionalities of anti-malware engines, malware detection and malware family labelling.

Our overarching observation is that anti-malware engines are over-reliant on using source-code-level strings for detecting and labelling a given binary. Our key observations can be summarized in the following points. They are also captured in figure 5.1.

a. Our simple evasion techniques work well. *MalEvasion* includes two simple evasion techniques but effective techniques achieving an evasion rate of more than 95% in our binaries. **Technique T1** manipulates strings in the source code (e.g. "Connection established") by adding a user provided string between every pair of characters in the

initial strings. **Technique T2** adds a line of code between every two lines of code in the source code. This line of code does not affect the functionality of the program.

Figure 5.1 summarizes the effect of applying our techniques on 560 MIPS binaries. Applying T1 leads to successful evasion from 22 engines to around 2 on average. Within two weeks, the engines learn and we have roughly 20 engines that classify the same modified binaries as malware. Applying T2 after makes the number of engines go down to roughly 5. T1 technique can also be reused with a new input string to evade detection.

b. Benign binaries can be misclassified by adding strings. We further demonstrate the engines's reliance on strings by causing false positives. We can do this by simply adding strings from a malware program to a benign program. We perform this by first manipulating strings found in a malware program. We compiled these programs and submitted the string-modified binaries to VirusTotal. Many engines including "high-reputation" engines like Avast, AVG and Fortinet detected these binaries as malware. One of the string-modified ARM binary and 5 of the MIPS binaries are classified as malware. Engines also use strings found in binaries for identifying the malware family. For example, for 81.7% of the binaries, an engine that gives a Gafgyt label to a malware binary also gives a Gafgyt label to the benign binary containing strings from that malware. Figures 5.6 and 5.7 illustrate these observations.

c. Engines that exhibit high recall on malware binaries are prone to false positives with benign binaries. We found that all the engines that achieve the top 10 recall rates (84.9% and above for our datasets) also more likely to misclassify benign binaries in which we added malware strings as shown in Tables 5.4 and 5.5.

5.1 Background and Datasets

In this section, we discuss about related background concepts and our datasets. We use the following definitions and metrics to evaluate the engines.

We use the term **corresponding binaries** to refer to different versions of the same program. For example, binaries in dataset D1 are compiled with -O0 option while binaries, while binaries in D2 are compiled with -O3 option. The performance of anti-malware engines will compare the classification and labels for such corresponding pairs.

a. Recall (E, D) is the percentage of binaries in a set of malicious binaries, D, that are flagged as malware by a set of engines, E.

b. Consistency (e, D1, D2). is the percentage of the pair of corresponding binaries across datasets D1 and D2 which receive the same label from a engine, e. Note that we can check consistency at levels of granularity: (a) identification of malware, and (b) the label of malware family.

Threshold for malware in VirusTotal,(T). How do we reach a consensus across the 71 engines in VirusTotal which do not always agree? Previous work suggests that a binary can be considered to be a malware if 2-15 engines flag it as malware [163]. 8.5 falls in the middle of this range. Hence we set the threshold, T to 8. For the purposes of our study, we will say that **the VirusTotal engines consider a binary as malware when more than 8 engines flag it as malware**. Note that varying this threshold will ultimately affect the trade off between false negatives and false positives.

We have used a total of 1750 in our study. Binaries in our dataset were compiled with the following compilation configuration. We did this because a malware author can also

Dataset	No. of Binaries
M1	280
M1_T1	280
M1_T1T2	280
M1_T1'	280
M2	240
M2_T1	240
B1	10
B1_MS	70
B1_OMS	70
Total	**1750**

Table 5.1: The binary datasets used in our study.

employ these techniques. This is because creating multiple binaries which are functionally equivalent but vary at the assembly code level can maximize the chance of evading detection. This can be easily achieved by using different compilation configurations [47, 27].

a. Architectures: We consider two architectures, ARM version 5 and MIPS R3000. We focus on these architectures because (a) they are vulnerable to IoT malware and (b) 66.0% of the ELF malware binaries belong to these architectures according to VirusTotal database [40].

b. Compilers and Versions: We consider four compilers and versions, GCCv5.5.0, Clangv9.0, Clangv4.0, and Clangv4.0 with OLLVM. For the remainder of the paper, we will use GCC to refer to GCC version 5.5.0, Clang to refer to Clang version 9.0, Clang4 to refer to Clangv4.0 and Clang4(OLLVM) to refer to Clangv4.0 used with OLLVM [**?**]. OLLVM is a tool that offers obfuscation options that can be used with Clang compilers. It offers 3 different kinds of obfuscations which can be applied during compilation. They are instruction substitution, bogus control flow and control flow flattening. We have used all the obfuscation options available in OLLVM in this study to investigate the effects of obfuscation on the VirusTotal engines.

c. Compilation optimization levels: We use five compiler optimization flags, O0,O1,O2,O3 and Os. We focus on stripped binaries in this work because around half of all ELF malware are stripped and malware analysis tools find stripped binaries more challenging [40, 59].

The data sets: Table5.1 summarizes our datasets. We have two main malware datasets, M1 and M2. M1 contains 280 malware binaries while M2 contains 240 malware binaries. Both datasets contain binaries that were compiled with the all the compilation configurations described above.

We made an effort to span a relatively wide range of malware types. We collected 8 of the malware programs from a GitHub malware repository, *threatland/TL-BOTS*, which contains source files of a vast array of malware families which includes Trojans, IRC, Mirai, Gafgyt, and QBots from 2014 to the present day. The remaining 5 codes were obtained from other online repositories in GitHub and Pastebin. Note that all the programs that we have used in our study have been used in recent studies on malware [59, 130].We use 7 of these programs to create the M1 dataset and the rest to create the M2 dataset. We will explain our choice of datasets in Section 5.4.

We also created a small main benign binary dataset, B1 containing 10 binaries. We obtained them by compiling a bubble sort program [**?**] using the GCC compiler with all five compilation options for both architectures. The bubble sort program only had 41 lines of code. All our source codes are written in the C language.

Creating modified malware datasets. To evaluate the engines, we apply our evasion techniques on our datasets, which we describe later. We name our datasets in the

following way. Names that do not contain the underscore symbol refer to main datasets. The part of the name before the underscore refers to the main datasets, M1, M2 or B1. The part of the name after the underscore refers to the evasion technique(s) that was applied on the source code to obtain the binaries in that dataset.

First, we created variations of M1 by applying: (a) evasion technique T1 on M1 to create M1_T1 with string parameter "PD", (b) technique T2 on M1_T1 to get M1_T1T2, (c) We also reapplied T1 on M1 with string parameter "*∧" to get M1_T1'. We also applied similar variations to the M2 dataset.

We generated B1_MS and B1_OMS in the following way. We extracted strings the *initial malware strings* (refer to as MS) from M1 and added them to B1 to form B1_MS. We extracted *our manipulated manipulated strings* (referred to as OMS) from M1_T1 and added them to B1 binaries to form B1_OMS. MS stands for malware strings and OMS stands for our manipulated strings. Binaries in B1_MS are called string-added binaries while binaries in B1_OMS are called string-modified binaries. In this work, we specifically define strings to be all characters found between a pair of unescaped double quotes found in source codes and binaries. A unique characteristic of IoT malware is that it constantly sends and receives messages from other infected devices and C&C servers. [**?**, 6]. These messages are found as strings in the source code.

5.2 Overview of *MalEvasion*

We present key ideas and insights in developing the *MalEvasion* framework. The key goal of *MalEvasion* framework is to provide techniques to evade detection from anti-

malware engines in VirusTotal. These techniques should be (a) easy to apply and (b) preserve original functionality of the malware program.

The insights that should not have worked. We present the insights that led to two simple methods that we thought that would never work. Much to our surprise, they did.

First, we thought of focusing on strings in the source code of the malware. These strings include the strings that appear in the communication between IoT malware and its C&C servers [6]. Upon investigation, we found that the source codes of IoT programs contain many strings that are used for such communication. We decided to investigate if manipulating these strings would be help the malware evade detection.

Second, for altering the sequence of commands, we thought of an equally simple approach that would not change the functionality of the code. We thought of inserting a single line of code, like a printf statement between every two lines of code in the source code.

Initially, we thought that the engines would consider the semantics of a binary code and sophisticated analysis of the structure of the binary. Hence, we thought that these ideas are way too simple to be effective. We were wrong.

We define our two evasion techniques, T1 and T2 below. We apply both techniques on malware source codes.

Technique T1 (param1): This function takes in a string parameter. We manipulate all the strings in the source code by adding this string parameter between every two characters in each string in the source code.

Ensuring no functional change. Note that we can easily remove the added characters of strings before we use the string. All we need is a simple function call that will remove the param1 string from between the original characters of the string. To prove this point, we add such function in the source code when we employ technique T1. This ensures that the resultant binaries from the modified program remains functionally equivalent to the original binary. In essence, this step ensures that the binary contains manipulated strings when it is statically analyzed. However, when the binary is actually executed, the original versions of the strings will be used.

This function takes any string of any length of the user's choice as its parameter. In this study, we used this technique with two kinds of string parameters, "PD" and "*∧". This serves to validate that this technique is effective when it is applied with different parameters. For simplicity, we use T1 to refer to cases where use use this function with "PD" as its parameter and T1' to refer to cases where we use use this function with "*∧".

Technique T2: In this technique, we add a non-functional statement between each two lines of code in the source code. Here we experiment with a *printf* with with null or an actual string. The method is general, we can choose to use it with a manipulated string or not. Even in the case where we have an actual string, the effect on the functionality of the malware is minimal. Furthermore, many IoT devices, like routers and printers do not come with a screen to display messages so the *printf* statement will have no actual effect.

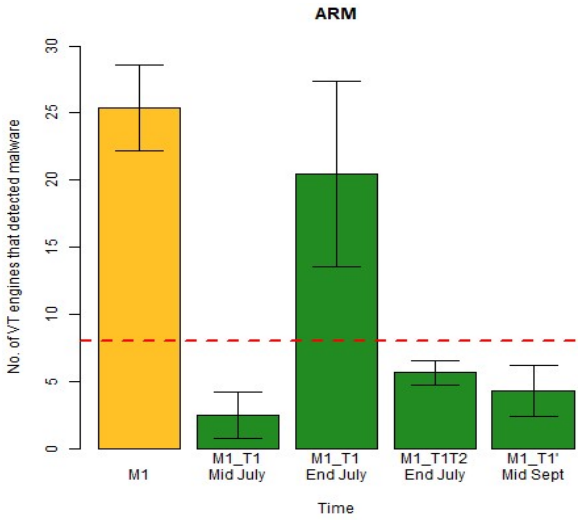


Figure 5.2: ***MalEvasion* can evade VirusTotal engines repeatedly and effectively for 100% of ARM the binaries in M1:** Applying T1 on ARM binaries in M1 reduces average number of engines that detect malwares from 25.4 to 2.49. Two weeks later, applying T2 on the binaries reduces average number of engines that detect malwares from 20.5 to 5.7. Applying T1' two months after we applied T1 reduces the engines that detected malware from 25.4 to 4.3.

5.3 Experimental Evaluation

We test the robustness of the engines by using *MalEvasion* and the datasets described in section 5.1. We group our experimental results around the following three questions. We answer each question with a series of observations.

Q1: Are our evasion techniques effective?

Q2: Can we make benign programs appear as malware?

Q3: Which engines are more reliable?

The following observation provides a summary of our study.

Overarching observation: Engines are overly reliant on strings for malware detection and labelling. For most engines, modifying source-code level strings leads to evasion causing false negatives. Adding strings found in malware binaries to benign binaries

results in being detected as malware, causing false positives. Binaries with similar strings also tend to share the same malware labels.

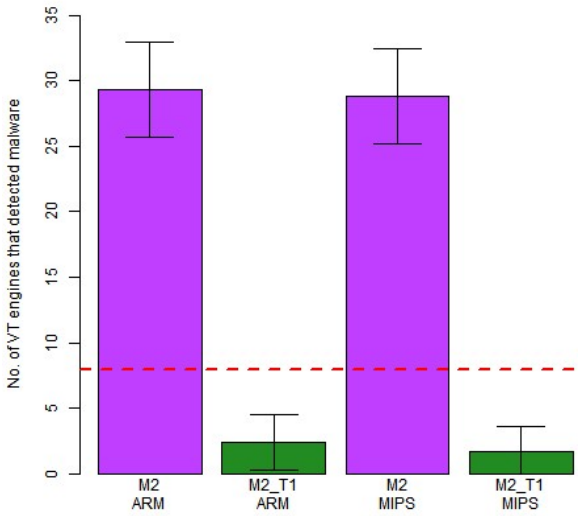


Figure 5.3: ***MalEvasion*** **evades VirusTotal engines for 100% of binaries in M2:** Applying T1 reduces the average number of engines that detect malwares from 29.3 to 2.5 for ARM binaries and from 28.8 to 1.7 for MIPS binaries in M2.

Q1: Are our evasion techniques effective? Our results show that both our evasion techniques, T1 and T2 can be applied to evade detection from engines for at least two months. The average and the standard deviation of the number of engines that detected the MIPS and ARM binaries in M1 as malware for each technique is shown in Figures 5.1 and 5.2. Figure 5.3 shows the results when T1 is applied on M2. Applying T1 on both main malware datasets, M1 and M2 causes evasion in 100% of binaries. Applying T2 on the M1 dataset causes evasion in 97.9% of ARM binaries and 99.3% of the MIPS binaries.

Observation 1: T1 achieves 100% evasion in M1 and M2. We follow the steps below to evaluate our evasion techniques.We first conduct the following multi-step study using the M1 dataset.We show the results for the MIPS and ARM binaries in Figures

5.1 and 5.2 respectively. Note that we repeat the first two steps with dataset M2, which corroborated our observations. Figure 5.3 shows our results for the M2 dataset.

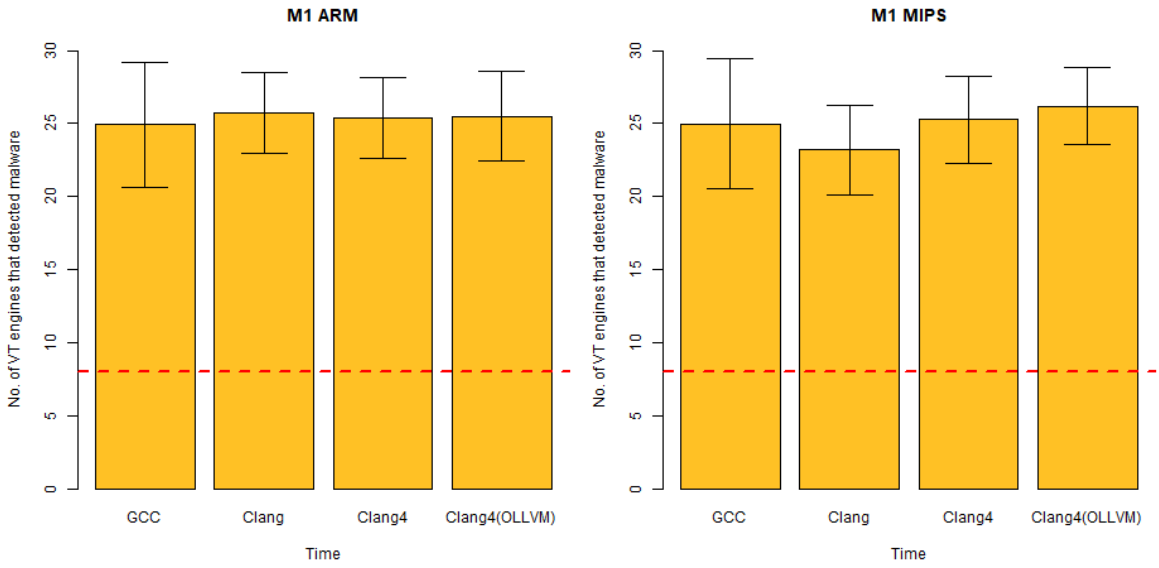


Figure 5.4: OLLVM obfuscation techniques do not lead to evasion: The number of engines that detected binaries as malware in M1 is between 25-26.5 for both architectures when Clang4 or Clang4(OLLVM) is used.

Step 1: Verifying the malicious nature of dataset M1. We find the number of engines that flag our binaries as malware. We confirm that more than T engines flag each of the binaries as malware. In each subsequent step, we observe the number of engines that flag the binaries as malware in the dataset used in that step.

In this step, we observed that using obfuscation techniques in the OLLVM tool, like instruction substitution, adding bogus control flow and control flow flattening on the binary does not lead to evasion from anti-malware engines. Using Clang4(OLLVM) does not reduce the average number of engines that flagged the binaries as malware. As shown in figure 5.4, the average number of engines that flagged binaries as malware in M1 is between

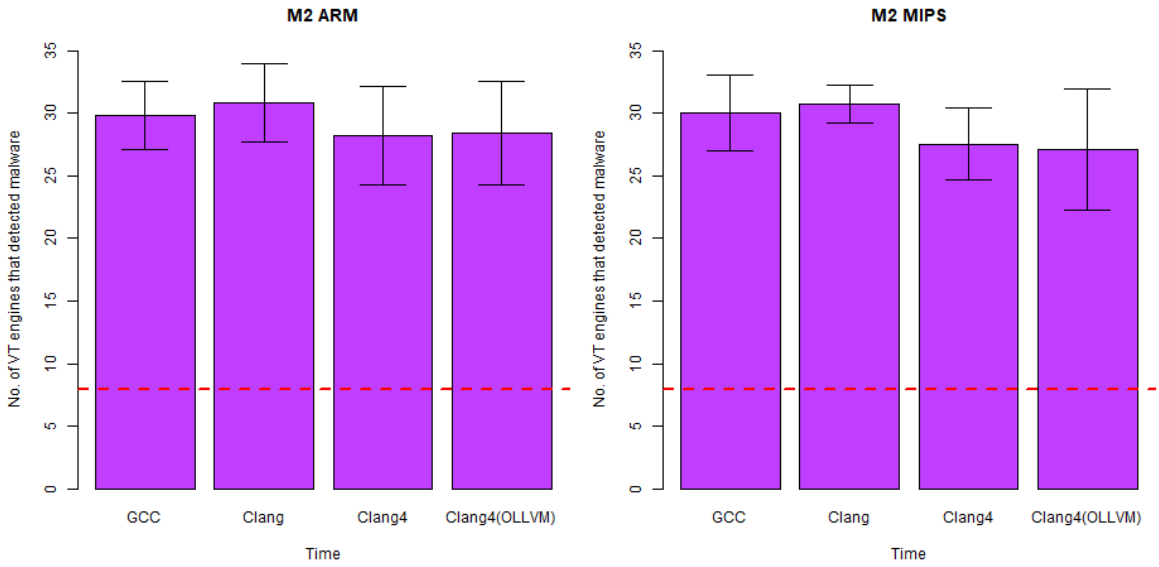


Figure 5.5: OLLVM obfuscation techniques do not lead to evasion: The number of engines that detected binaries as malware in M2 is between 27-28.5 for both architectures when Clang4 or Clang4(OLLVM) is used.

25 and 26.5 for both architectures when Clang4 or Clang4(OLLVM) is used. As shown in figure 5.5, the average number of engines that flagged binaries as malware in M2 is between 27 and 28.5 for both architectures when Clang4 or Clang4(OLLVM) is used.

Step 2: We assess engines using the M1_T1 dataset. Recall that M1_T1 is created by applying technique T1 on the M1 dataset.

Step 3: We reassess engines using the M1_T1 dataset two weeks later. The goal is to detect how the engines evolve.

Step 4: We assess engines using the M1_T1T2 dataset. We obtain M1_T1T2 by applying technique T2 on M1_T1.

Step 5: We assess engines using the M1_T1' dataset 2 months after Step 2. We then compare the results.

Technique T1 can be used to evade engines for binaries of all compiler configurations. The average number of engines that detected the ARM and the MIPS binaries as malware in dataset M1 is 25.4 and 24.9 respectively. The standard deviation is 3.2 and 3.5 respectively. Applying T1 on M1 resulted in 100% evasion for binaries belonging to both architectures. Figures 5.1 and 5.2 show that the average number of engines that detected the MIPS and the ARM binaries as malware in M1_T1 is 2.6 and 2.5 respectively. The standard deviation is 1.5 and 1.7 respectively.

Applying technique T1 on dataset M2 gives similar results. Figure 5.3 shows that the average number of engines that detected the initial ARM and the MIPS binaries in M2 is 29.3 and 28.8 respectively. The standard deviation is 3.7 and 3.6 respectively. Applying T1 on M2 resulted in 100% evasion for both architectures. The average number of engines that detected the ARM and the MIPS malware in M2_T1 is 2.5 and 2.6 respectively. The standard deviation is 1.7 and 1.5 respectively.

Observation 2: Engines learn to detect binaries in M1_T1 as malware in two weeks. We were pleasantly surprised to see that VirusTotal engines seemed to learn: they were able to detect 99.3% of ARM and MIPS binaries in M1_T1 as malware 2 weeks after we submitted them to VirusTotal. The average number of engines that detected the ARM and the MIPS binaries as malware is 20.5 and 20.9. The standard deviation for both architectures is 6.9. This observation is an indication that engines are updated frequently which aligns with observations from previous studies [163].

Unfortunately, as we will see in later observations, at least a part of this learning is only "string-deep", which means that: (a) insufficient to provide a permanent solution to our T1 technique, which can change its input string parameter.

Observation 3: Technique T2 achieves greater than 95% evasion. We apply technique T2 on M1_T1 and obtain dataset M1_T1T2. As shown in Figures 5.1 and 5.2, we find that 99.3% of the MIPS binaries and 97.9% of ARM binaries evaded detection by the engines. The average number of engines that detected the MIPS and the ARM binaries in M1_T1T2 as malware is 5.8 and 5.7 respectively which is below the detection threshold of 8. The standard deviation is 1.0 and 0.9 respectively.

Observation 4: Technique T1 can be effective repeatedly: just use a new string parameter. We apply technique T1 with a new string parameter "*∧" to create dataset M1_T1': 100% of its binaries evade detection! As shown in Figures 5.1 and 5.2, the average number of engines that detected the binaries in M1_T1' is 3.5 and 4.3 for MIPS and ARM binaries respectively. The standard deviation is 2.2 and 1.9 respectively.

What makes this observation more interesting is that by now the engines have been trained on datasets M1_T1 and M1_T1T2. All binaries in M1_T1T2 are classified as malware 1.5 months after the they were submitted to VirusTotal. The average number of engines that detected the ARM and the MIPS binaries as malware in this dataset is 23.0 and 23.4 respectively. The standard deviation for both architectures is 2.5.

Observation 5: The engines rely heavily on strings for detection. Summarizing these four observations, we can state that engines use strings to detect malware. Observation 1 shows that applying technique T1 provides very high evasion rates for both

architectures. Observation 4 shows that technique T1 can be repeatedly applied to the malware source code, each time with different parameters to achieve good evasion even after 2 months since we first applied and submitted binaries obfuscated with T1.

Q2: Can we make benign programs appear as malware? Here we want to go deeper and understand in more detail what features trigger detection by the anti-malware engines. To achieve this, we consider two complementary approaches.

First, we analyze the percentage of binaries that were classified as malware before and after applying our evasion techniques. A decrease in in the number of engines that detected a malware binary as malware indicates that our evasion techniques have modified some feature in the binary that is being used by engines to detect malware. Observations 1 - 4 suggest that engines use: (a) source code level strings, as shown by technique T1, and (b) sequence of instructions, as shown by technique T2.

Second, we can reduce the instructions in order to isolate the instructions that trigger detection by the engines. For this, we remove parts of a source code before compiling it and submitting it to VirusTotal. We find that a small set of source-code strings is enough for engines to detect the malware. We use this technique to make observations 6 and 8.

Observation 6: Engines misclassify benign programs if they have appropriate strings. We added all strings found in each malware program in M1 separately to the bubble sort program described in Section 5.1 and compiled them to form the B1_MS dataset. In more detail, the strings from each of the malware program were added into a string array in the sort program. A for loop was added to print all the strings in the array in the main function of the program. This loop ensures that all the strings in the array is

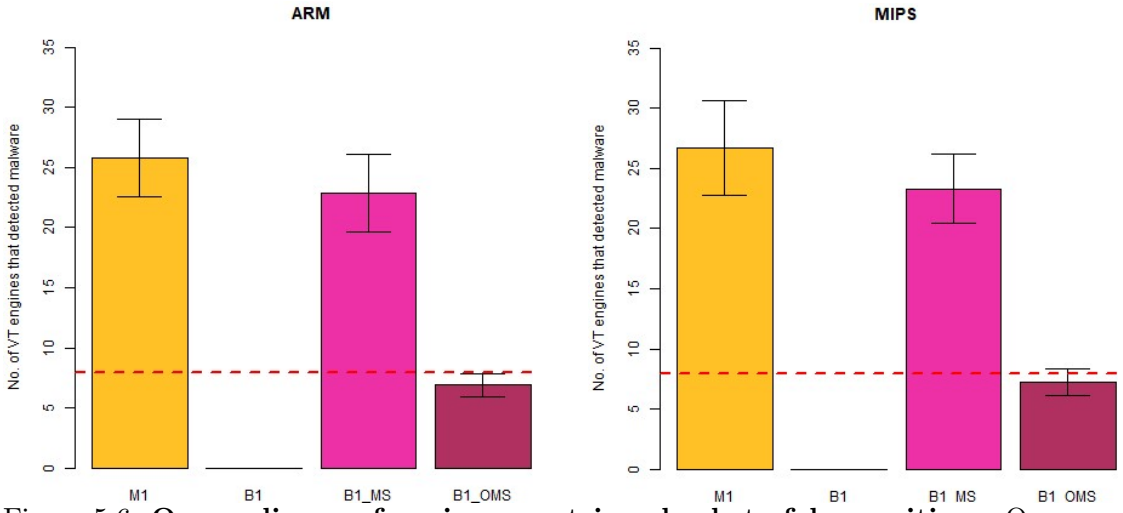


Figure 5.6: **Over-reliance of engines on strings leads to false positives:** On average, 22.9 and 6.9 engines detected ARM binaries in B1_MS and B1_OMS as malware.On average, 23.3 and 7.2 engines detected MIPS binaries in B1_MS and B1_OMS as malware.

used in the program. This step prevents the situation where unused variables get omitted during the compilation process. We compiled all the modified sorting program with GCC for both architectures for all compilation optimization levels. Note that the unmodified sort program was not detected as malware by any of the engines for both architectures for all optimizations.

Figure 5.6 shows that adding malware strings to a benign sorted program increases the average number of engines that detected the ARM and MIPS binaries from 0 to 22.9 and 0 to 23.3 respectively. The standard deviation for the modified sorted programs is 3.2 and 2.9 for the ARM and MIPS binaries respectively. For reference, this roughly the number of engines that detected the binaries compiled with GCC in M1 (25.8 for ARM and 26.7 MIPS). As a result, all binaries in B1_MS are classified as malware as they have significantly more than 8 engines vouching against them.

Set of strings	Misclassifying Engines	
(added to benign programs)	ARM binaries	MIPS binaries
"GET /fuck1hex"	Kaspersky	-
"buf: %s;", "/proc/cpuinfo" , "gethostbyname", "BOGOMIPS", "assword:", "ncorrect"	Avast, AVG, Avast-Mobile, Kaspersky, ZoneAlarm by Check Point	Avast-Mobile, Kaspersky
"root", "invalid", "incorrect","user", "login", "name", "gayfgt","buf: %s", "/bin/sh", "/proc/cpuinfo", "BOGOMIPS"	Avast, Avast-Mobile, AVG,Rising Sophos	Avast-Mobile, Rising, Sophos

Table 5.2: **Engines classify benign programs with strings from malware as malware:** Sets of malware strings whose appearance in benign programs leads to misclassification and the engines that are fooled. Surprisingly, the misclassification is not consistent across the ARM and MIPS architectures.

How far can push the misclassification of benign binaries? Intrigued by the above results, we want to push the envelope further. We find that only a small subset of malware strings can lead to a misclassification. Each row in Table 5.2 shows a sets of strings used by some engines to detect malware. The occurrence of these strings in the modified sort binary is sufficient to make the engines flag it as malware. Surprisingly, the engines that detect an ARM and a MIPS binary containing these sets of strings and the instructions for the sort program as malware is different in some cases. Although we have the full list of strings that can cause many engines to detect malware, we cannot show all of them due to page limit restrictions.

Taking one step further, we show that some engines seem to solely rely on identifying signature strings to detect malware. We ascertain this by compiling a C program that only contains a set strings shown in Table 5.2 and a for loop to print these strings. This short program of around 10 lines make the engines in the table flag them as malware. This suggests that at least some engines can classify a program based solely on blacklisted strings.

Observation 7: Engines rely on strings for malware family classification. Here we go deeper into the misclassification of string-enhanced benign binaries and examine what malware family is reported. We find that the malware family label of 83.1% and 79.3% of the string-enhanced ARM and MIPS benign binaries in B1_MS is the same as the corresponding malware binary in M1. This observation seems to further validate that the engines rely heavily on strings for malware family classification.

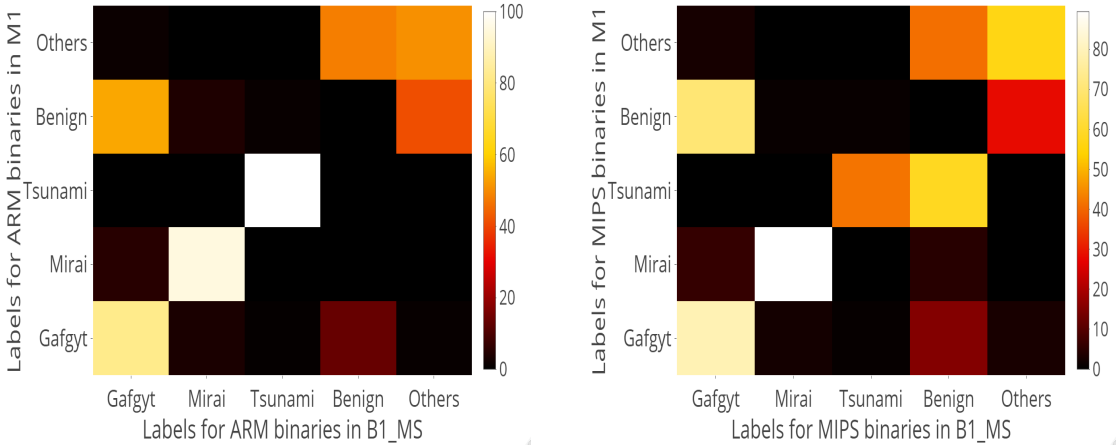


Figure 5.7: **Engines use strings to label malware:** For ARM binaries, the bright diagonal grids for Gafgyt, Mirai and Tsunami show that when the malware from M1 is classified as belonging to one of these families, greater than 80% of the corresponding benign binary will be assigned to the same family. For MIPS binaries, the bright diagonal grids for Gafgyt and Mirai show that when the malware from M1 is classified as belonging to one of these families, greater than 75% of the corresponding benign binary will be assigned to the same family.

Note that we focus on three major IoT malware families: Gafgyt, Mirai and Tsunami [40]. For simplicity, we did not distinguish between different sub-classifications or versions in a family. For example, we consider a binary to belong to the Gafgyt family, if the assigned label contains the family name, "Gafgyt". Note that since we only compare labels given by a specific engine, the fact that different engines may have different labelling conventions [68, 99] does not affect our observations here.

In more detailed, a heatmap-style matrix that captures the full picture of label similarity is shown in Figure 5.7. The bright diagonal grid for Gafgyt and Mirai in both figures show that when the malware binary is classified as Gafgyt or Mirai, greater than 75% of the corresponding benign binary also receives the same label. For ARM binaries, when the malware binary is classified as Tsunami 100%, of the corresponding benign binary also receives the same label. For MIPS binaries, this percentage is 41.7%. Additionally,

we also found that 32.8% and 25.8% of the corresponding pairs for the ARM and MIPS binaries received identical labels.

Strings	Triggered Engines
Initial strings: "Failed opening raw socket.","SCANNER ON — OFF", "OFF" **Our strings:** "FPDaPDiPDlPDePDdPD PDoPDpPDePDnPDiPDnPDgPD PDrPDaPDwPD PDsPDoPDcPDkPDePDtPD.", "SPDCPDAPDNPDNPDEPDRPD PDOPDNPD PD—PD PDOPDFPDF", "OPDFPDF"	AhnLab-V3, Fortinet
Initial string: "cd /tmp —— cd /var/run ——cd /mnt —— cd /root —— cd/;busybox tftp 185.112.248.68 -c get tftp.sh;sh tftp.sh;busybox tftp -r tftp2.sh -g185.112.248.68; shtftp2.sh;busybox wgethttp://185.112.248.68/gtop.sh;chmod +xgtop.sh; sh gtop.sh" **Our string:** "cPDdPD PD/PDtPDmPDpPD PD—PD—PD PDcPDdPD PD/PDvPDaPDrPD/PDrPDuPDnPD ..."	Ikarus
Initial string: "(null)" **Our string:** "(PDnPDuPDlPDlPD)"	Fortinet

Table 5.3: **Engines add new strings found in malwares as signatures:** the strings that we have created eventually become "signatures" for malware. Adding these strings in benign software makes some engines flag them as malware!

Observation 8: Engines add newly found malware strings to their set of features. Some engines detect binaries in B1_OMS as malware. Figure 5.6 shows the average number of engines that detected the ARM and the MIPS binaries in B1_OMS is 6.9 and 7.2 respectively. The corresponding standard deviation is 0.9 and 1.1. One of the ARM binary and five of the MIPS binaries were classified as malware because more than 8 engines detected them as malware.

Similar to our findings in observation 6, only a small subset of strings from M1_T1 is required to be found in benign binaries to be detected as malware. Table 5.3 shows some of the strings from M1_T1 that are used by the engines to detect malware. For ease of reading, we have also stated the original version of the strings. Binaries containing these strings were detected as malware by the same engines regardless of the architecture of the binary. Interestingly, we note that the presence of corresponding original versions of these strings in benign binaries do not cause any engines to detect it as malware.

This observation shows that malware authors can easily influence the set of black-listed strings that are used by engines for malware detection. This opens up the possibility

ARM		MIPS	
B1_MS	B1_OMS	B1_MS	B1_OMS
ALYac,AVG, Ad-Aware, AhnLab-V3, Avast,Avast-Mobile, BitDefender, BitDefender-Theta, ClamAV,Emsisoft, FireEye, Fortinet, GData,Kaspersky, MAX, MicroWorld-eScan, Microsoft,Rising, Sophos,Tencent, ZoneAlarm	AVG, AhnLab-V3, Avast, Avast-Mobile, Cyren,Fortinet	ALYac,AVG, Ad-Aware, AhnLab-V3, Avast, Avast-Mobile, BitDefender, BitDefender-Theta, ClamAV,Emsisoft, FireEye,Fortinet, GData,Kaspersky, MAX, MicroWorld-eScan, Microsoft,Rising, Sophos,Symantec, Tencent, ZoneAlarm	AVG, AhnLab-V3, Avast, Avast-Mobile, Cyren, Fortinet

Table 5.4: **The overly aggressive engines:** we list the engines that classify at least 80% of the benign binaries in B1_MS and B1_OMS as malware.

for the malware author to add strings that are commonly found in benign programs into malware. If these engines use any of these strings to detect malware, then benign ware containing the string will be detected as malware. Malware authors can also inject some of the strings from malware into benign ware to make it seem like malware. These actions will reduce the credibility of these engines.

Q3: Which engines are more reliable? We have seen that some engines tend to be over reliant on using string features to detect malware. The logical next step is to identify which engines are more reliable for which compilation configurations. In practice,

we are unaware of the compilation configurations that are used to produce the given binary.

We quantify the reliability of an engine by using two metrics, recall and consistency.

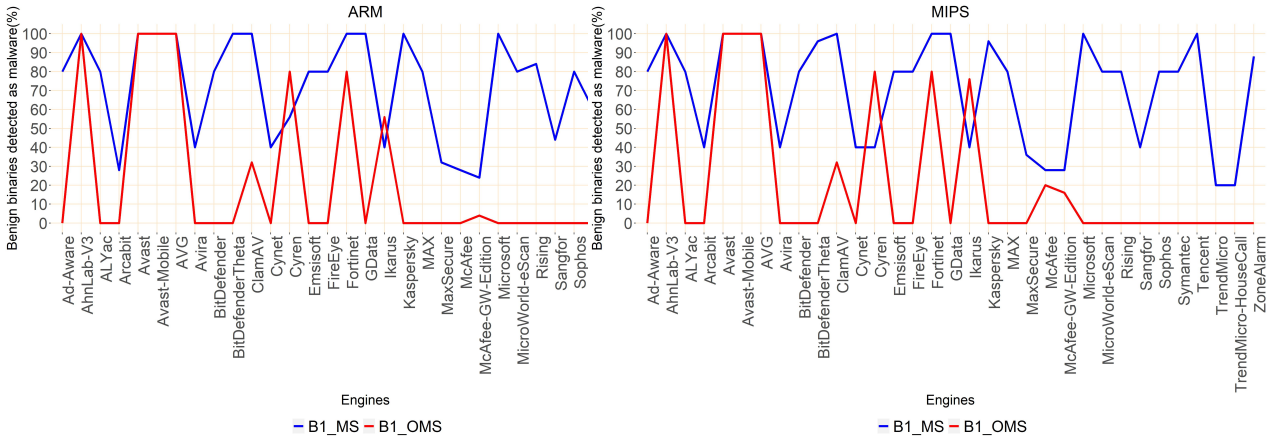


Figure 5.8: False positive detection: Many Engines detected benign binaries: 33 engines detect benign binaries with malware strings as malware

Observation 9: 21 engines incorrectly detect at least 80% of binaries in B1_MS. Figure 5.8 shows all the engines that detected benign binaries as malware. Table 5.4 shows the engines that detected atleast 80% of the binaries in B1_MS as malware. 21 of these engines incorrectly detected 80% or more of binaries in B1_MS belonging to both architectures. We observed similar behaviour from one more engine, Symantec for MIPS architecture.

A lesser number of engines detected binaries in B1_OMS as malware compared to the B1_MS dataset. 6 engines incorrectly detected 80% or more of the binaries in B1_OMS for both architectures. We conjecture that a lesser number of engines detect the benign binaries in B1_OMS because the manipulated strings found in malware was introduced more recently by us in July. Hence, only a few engines have added our obfuscated malware strings as a feature to detect malware.

ARM		MIPS	
Recall	Consistency	Recall	Consistency
Avast-Mobile AVG Avast Fortinet TrendMicro- HouseCall, Ikarus, GData, Microsoft, MicroWorld- eScan, BitDefender, MAX,Ad-Aware	Avast-Mobile, Fortinet, DrWeb, AVG, Avast, Ikarus, TrendMicro, -HouseCall, McAfee-GW- Edition, McAfee, BitDefender, Ad-Aware, MicroWorld- eScan, MAX,FireEye	Avast-Mobile, Avast,AVG, TrendMicro- HouseCall, Ikarus,Fortinet, GData,Microsoft, MicroWorld- eScan, Ad-Aware,MAX	Fortinet, Avast-Mobile, Ikarus, AVG, Avast, TrendMicro- HouseCall, McAfee, McAfee-GW- Edition, MicroWorld- eScan, Ad- Aware,FireEye, BitDefender, MAX,Emsisoft

Table 5.5: **Top engines:** We show the top performing engines that give the top 10 scores for recall and consistency for each architecture.

Observation 10: No-free-lunch: The higher-recall engines on malware datasets exhibit higher misclassifation for benign datasets, B1_MS and B1_OMS

Table 5.5 shows the best performing engines that gives the top 10 scores for recall and consistency. Here, we calculate the performance of all engines in VirusTotal for 4 malware datasets, M1, M2, M1_T1 and M1_T1T2. Note that we recorded these results 2 weeks after we submitted the binaries in M1_T1 and M1_T1T2 to VirusTotal. The top engines give a recall score of 84.4% or higher for ARM binaries and 85.3% or higher for MIPS binaries.

We calculate the consistency by finding the average consistency between two groups of malware binaries. First, we find the percentage of corresponding binaries that were detected as malware for: (a) M1 and M1_T1 datasets, and (b) M1 and M1_T1T2 datasets. The top engines give an average consistency score of 84.2% or higher for ARM binaries and 84.7% or higher for MIPS binaries.

Interestingly, we found that all the list of top engines, except the DrWeb engine, have misclassified some benign binaries in B1_MS. All except four engines in this list have misclassified 80% or more of the binaries in B1_MS. The better performing four engines, TrendMicro-HouseCall, Ikarus, McAfee and McAfee-GW-Edition misclassified 20-75% of the binaries in B1_MS.

Observation 11: Technique T1 affects malware family-level labelling consistency. We study the malware labels given by each engine to each pair of corresponding binaries in the M1 and M1_T1 datasets. Surprisingly, we found that in most cases, each binary in the pair is assigned to different malware families. Figure 5.9 summarize the results for ARM and MIPS binaries respectively in a heatmap-style matrix.

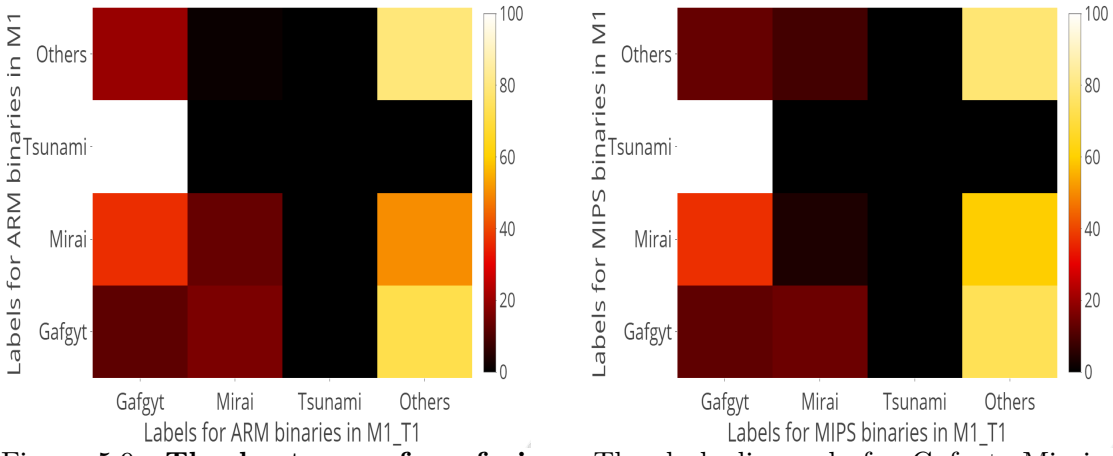


Figure 5.9: **The heatmap of confusion:** The dark diagonals for Gafgyt, Mirai and Tsunami show that less than 20% of the corresponding pairs of ARM binaries from the M1 and M1_T1 datasets are assigned to the same malware family. The dark diagonal cells for Gafgyt, Mirai and Tsunami show that less than 20% of the corresponding pairs of MIPS binaries from the M1 and M1_T1 datasets are assigned to the same malware family. Furthermore, the white cell (Gafgyt, Tsunami) means that 100% of the binaries initially detected as Tsunami are classified as Gafgyt after applying technique T1.

The dark diagonal grids for Gafgyt, Mirai and Tsunami in both figures show that for both architectures, when the malware binary from the M1 dataset is classified as Gafgyt,

Mirai or Tsunami, less that 20% of the corresponding binary in the M1_T1 dataset is given the same malware family label. In more detail, for both architectures, when the malware binary from M1 is classified as a Gafgyt sample, 70-75% of the corresponding binary receives a family label which does not represent any of the three families. This percentage becomes 50-60% when we consider Mirai samples.

We also found that only 2.6% and 2.5% of the corresponding pairs for the ARM and MIPS binaries received identical labels. As we have shown in observation 7, a much higher percentage, 32.8% and 25.8% of the corresponding pairs for the ARM and MIPS binaries received identical labels when we compare M1 and B1_MS. This finding suggests that the labelling functionality in engines mainly relies on strings to decide the family of a malware.

	ARM			**MIPS**		
Engines	**M1**	**M1_ T1**	**M1_ T1T2**	**M1**	**M1_ T1**	**M1_ T1T2**
Kaspersky	99.3	0	0	100	0	0
Sophos	75.0	0	0	70.7	0	0
ESET-NOD32	99.3	72.1	0	97.1	77.1	0

Table 5.6: **Even high-reputation engines have poor recall on modified malware:** the engines have high recall for the initial malware but poor recall for malware manipulated by *MalEvasion*.

Observation 12: String manipulation affects recall rates of even for "high-reputation" engines. Current literature and study has collectively and informally suggested a group of "high-reputation" engines[1] [26, 145, 79, 163]. However, even these engines are affected by our simple evasion techniques by often a significant drop in their detection: from 70-99% down to 0!. Table 5.6 summarizes the results for some of these

[1]The engines that are consistently reported as better performing are: Kaspersky, Symantec, AVG, F-Secure, Ikarus, McAfee, Microsoft, ESET-NOD32, and Sophos.

engines. Kaspersky gives almost perfect recall for binaries in M1 for both architecture, but gives 0% recall for binaries in M1_T1 and M1_T1T2. Sophos has a 70 - 75% recall for binaries in M1 but gives 0% recall for corresponding binaries in M1_T1 and M1_T1T2. ESET-NOD32 has atleast 97% recall for binaries in M1, and drops only to 72-77% with technique T1, and drops to zero when both techniques are applied. The significant reduction in recall for binaries in M1_T1T2 compared to binaries in M1_T1 and M1 could be attributed to a significant reliance on string-based and command-sequence fingerprinting patterns even for these higher-reputation engines.

The difference in recalls between the M1 and the M1_T1 dataset is atleast 20% for all high-reputation engines except AVG and F-Secure for both architectures. AVG gives a recall of atleast 95% for both datasets for both architectures. F-Secure consistently gives a recall of below 10% for both architectures for all four malware datasets, M1, M1_T1, M1_T1T2 and M2. Hence, the difference in its recall scores for the M1 and M1_T1 datasets is also minimal. Three engines, Ikarus, McAfee and Symantec give higher recalls for the ARM binaries in M1_T1 dataset compared to M1. The difference in recalls are 22.1%, 55.0% and 32.14%. The corresponding difference in recall for MIPS binaries are 25.7%, 52.9% and 34.3%. Microsoft performs better for the M1 dataset by 26.4% compared to the M1_T1 dataset for recall. The corresponding difference for MIPS is 25%.

Overall, our observation shows that with the exception of AVG, none of the other high-reputation engines give consistently high recalls for M1 and M1_T1 datasets. This shows that there there is a good chance for both versions of the malware author's binary,

the one compiled without using any of our evasion technique and the one compiled after applying technique T1 to evade few of the high-reputation engines.

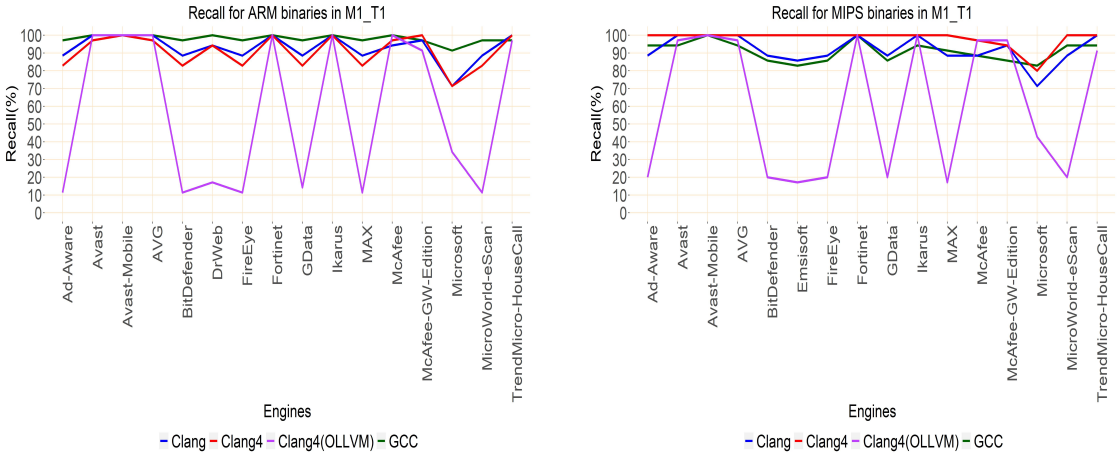


Figure 5.10: The recall of the engines is particularly low for the Clang4(OLLVM) compiler in M1_T1 for both ARM (left) and MIPS (right).

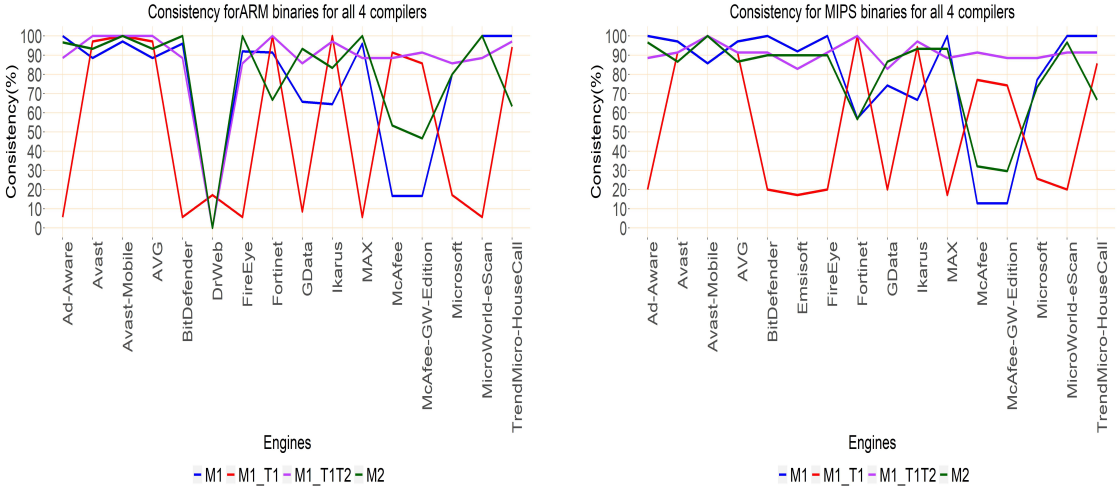


Figure 5.11: The M1_T1 dataset exhibits the least consistency for compilers among all the datasets.

Observation 13 (Sensitivity 1): The choice of compiler has significant effect on malware detection rates. We study how the compiler affects the performance of the engines. Here, we present results for the top engines mentioned in Observation 9.

We find that the recall and consistency for each of these engines for both metrics for each of the four malware datasets, M1, M1_T1, M1_T1T2 and M2 separately.

Using technique T1 and Clang4(OLLVM) increases the chan- ces of evasion. We want to find the dataset and the compiler that would give the worst average recall and consistency. To do this, we take the average recall for all top engines for binaries compiled with each compiler for each dataset. We find that the engines give the lowest recall in the M1_T1 dataset for binaries compiled with Clang4(OLLVM). The average recall for binaries in M1_T1 compiled with Clang4(OLLVM) is 57.0% for ARM and 59.8% MIPS. The recall rates given by the other three compilers for this dataset for both architectures is above 90%.

Upon farther investigation, we find that that half of the 16 top engines for each architecture exhibit poor recall: 20% or less for both architectures. Figure 5.10 illustrates this result. We find that the remaining 8 engines, Avast-Mobile, Fortinet, Ikarus, Avast, AVG, TrendMicro-HouseCall,McAfee-GW-Edition and McAfee give recalls of 85% and above for all compilers for this dataset.

Dataset M1_T1 also gave the lowest average consistency among all the datasets. In this section, we are considering the consistency in detection between 4 binaries compiled with 4 different compilers, GCC Clang, Clang4 and Clang4(OLLVM) when all other compiler configuration are kept constant. The average consistency for all engines in this dataset for ARM binaries is 52.3% while the average consistency for MIPS binaries is 54.6%. This result is not surprising, because half of the top engines had a low recall of 20% or less for binaries compiled with Clang4(OLLVM) in this dataset. These engines, except Microsoft,

also have a consistency score of 20% or below for architectures. Microsoft has a consistency score of 17.1% and 25.7% for ARM and MIPS binaries. Figure 5.11 shows the consistency of each of the top engines for all four malware datasets for both architectures. The average consistency for all the other datasets for both architecture is at least 74%.

A malware author can maximise her chances of evasion by compiling his binaries with multiple compilers. Interestingly, we observe that only 3 engines give consistently high consistency scores for all datasets for both compilers. These three engines are Avast-Mobile, AVG, and Avast and they give a consistency score of at least 85% for all datasets for both architectures. We do not consider DrWeb because it gives 0% recall for ARM binaries compiled with Clang4 and Clang4(OLLVM) for M1 and M2. All other engines have a minimum difference of at least 30% between consistency scores of the four datasets for both architectures. This shows that these engines have a difficulty in detecting all the four versions of a binary that was compiled with the four different compilers in at least one dataset.

Observation 14 (Sensitivity 2): Compilation options have a relative small effect on the performance of the engines. Here we present the results for binaries in M1. Three engines, Ikarus, McAfee and McAfee-GW-Edition have at least 15% higher recalls for binaries compiled with O0 options compared to binaries compiled with O1 and O2 options in M1. DrWeb gives a recall that is 17.9% lower for binaries compiled with the Os option compared to other options. We did not observe any difference above 15% in recall rates between binaries compiled with various optimizations for the other datasets.

5.4 Discussion

We discuss the broader context and limitations of our work.

a. How will *MalEvasion* be used in practice? *MalEvasion* can be used by the analysts and engineers to stress-test the effectiveness of an anti-malware engine. Specifically, our approach can test in an engine is able to: (a) detect both the initial and the manipulated versions of a malware binary, (b) detect malware binaries that are compiled with various compilers and compilation options as malware, (c) prone to false positives, and (d) can correctly identify the malware family label consistently across manipulations and different compilation configurations.

b. Are our datasets representative? This is a difficult question for any empirical study, that can mostly be answered indirectly. First, recall that our malware covers several major malware families, such as Gafgyt, Tsunami, and Mirai, as we discussed in a few places earlier. Second, our malware is real malware because not only it is described as such, but the combined wisdom of the engines classifies it as such in its initial version. Third, recent studies have found significant number of malware source code on public software archives like GitHub [121, 80, 95, 6]. These studies argue that malware authors tend to make their codes publicly available for two reasons: (a) for creating an online brand and reputation, and (b) for establishing an alibi if their code is found implicated in an attack. With public source code, they can hide behind the excuse that many people had access to the code when they face legal issues [80]. As explained earlier, we collected all our malware source codes from popular public software repositories that contain a wide variety of

malware source codes. Source codes from these repositories have also been used in recent malware related works [59, 130].

c. How powerful are our evasion techniques? A key message from our study is that simple techniques cause significant evasion on state of the art detection engines. Furthermore, the goal of our work is to investigate how robust anti-malware engines are against IoT malware from the perspective of the malware author. Our simple evasion techniques, especially T1 leverages the fact that IoT malware contains many strings that are used for C&C communication. In section 5.3, we show that both our techniques can give high evasion rates of above 95% for binaries compiled with various architectures and compilation configurations. We also provide evidence that our evasion techniques work because engines rely heavily on string features to detect and label malware. We validated this by submitting benign binaries containing malware strings to the VirusTotal engines. Hence, our evasion techniques will work well as long as two requirements are met: (a) the malware program contains strings, and (b) the engine analyzing the malware places high importance on string features, just like most of the VirusTotal engines seem to do now.

d. Do these results apply to the desktop versions of engines? Some of these engines in VirusTotal have desktop versions, meaning the full commercial software version that a security analyst can download and use to analyze IoT malware. Previous work compared the detection capability of the desktop version to the version in VirusTotal and found that the VirusTotal versions have higher F1 scores [163]. Based on this, one could argue that the VirusTotal versions could be reasonable indicators of the capability

of the desktop versions. A more authoritative answer would require some experimentation, which extends beyond the scope of this paper.

5.5 Related Work

To the best of our knowledge, there has not been any previous study that focuses on applying simple evasion techniques on source codes to evade and evaluate VirusTotal engines for ARM and MIPS binaries. Furthermore, there is relatively limited work done at the intersection of (a) IoT malware, (b) VirusTotal engines evasion via source code manipulation and (c) VirusTotal engines evaluation.

We group related previous work into the categories below.

a. Modifying source-code to evade detection: Very few works have focused on this flavor of the problem. The most relevant work modified Visual Basic (VBA) source-code programs on Windows machines to avoid detection from 3 anti-malware engines [32].

b. Modifying binaries to evade detection: A large body of work focus on modifying malware *binaries* to avoid detection. This is a significantly different problem from what we focus on here. The majority of the work focuses on Windows platforms in contrast to our focus on IoT platforms [7, 30, 88, 43, 61]

c. Evaluating anti-malware engines on VirusTotal: We found two relatively recent works which study the performance of engines on VirusTotal. The most recent work assesses the stability of the classification of VirusTotal engines for over a year [163] but did not do any code manipulation. Their key result was that VirusTotal engines tend to detect obfuscated benign binaries as malware. Another work engines [99] found that on average,

VirusTotal engines have a recall of 59% and finds the malware family correctly for only 6.4% of the binaries. Neither of these studies proposes evasion methods, as we do here, and this enables us to understand deeper what features confuse these engines.

5.6 Conclusion

In this work, we put ourselves in the shoes of a malware author and ask the question: "how can we avoid detection with minimal effort?" By doing so, we also stress-test the limits of the capabilities of anti-malware engines. We develop *MalEvasion*, a systematic framework to stress-test and confuse VirusTotal engines, which can help us understand what features these engines focus on.

First, we find that our two simple techniques, string manipulations and adding "filler" lines of code is sufficient to give high evasion rates of above 95%. These solutions require minimal effort to be deployed and they do not change the functionality of the program.

Second, we find that engines place high importance on strings found in binaries, causing them to report false positives. We do this by submitting a program containing our string to VirusTotal. VirusTotal engines learn to use our strings in malware binaries as malware signatures. Adding these strings to a benign program will cause it to be detected as malware.

Third, we find that engines that give high recall and consistency also produce false positives.

We believe that our study is a significant step in understanding how VirusTotal engines detect and label malware. We envision that our framework will be used to improve the detection and labelling capabilities of VirusTotal engines.

Chapter 6

Conclusion

The key problems that we address in this thesis are: (a) which disassembler should a malware analyst choose to get the most accurate disassembly to detect, analyze and defuse IoT malware quickly, and (b) how easy it is for IoT malware authors to evade anti-malware engines? Our studies show that: (a) none of the existing dis assemblers can give accurate disassembly for binaries of various architectures and compilation configurations and that combining disassemblers gives the most accurate disassembly, and (b) anti-malware engines can be easily evaded by using simple techniques to modify malware source code.

DisCo provides a solution to the problem of inaccurate disassembly because it combines the collective power of disassemblers effectively. We showed that it consistently outperforms the best disassembler in a group of disassemblers. For example, our approach outperforms the best contributing disassembler by as much as 17.8% for F1 score for function start identification for MIPS binaries compiled with GCC with O3 option. We then show that the collective power of the disassemblers can be brought back to improve each

disassembler. We showcase this capability by developing Ghidra+, which outperforms the initial Ghidra by as much as 13.6% in terms of F1 score by simply using function signatures identified in our approach. In addition, our systematic evaluation within our approach led to a bug discovery: a bug introduced in Ghidra 9.1, for which the Ghidra team expressed appreciation.

We also found that anti-malware can be easily evaded for months by applying simple evasion techniques in the source code. We develop MalEvasion, a systematic framework to stress-test and confuse VirusTotal engines, which can help us understand what features these engines focus on. First, we find that our two simple techniques, string manipulations and adding "filler" lines of code is sufficient to give high evasion rates of above 95%. These functionality preserving solutions require minimal effort to be deployed. Second, we find that engines place high importance on strings found in binaries, causing them to report false positives. We do this by submitting a malware program containing our strings to VirusTotal. VirusTotal engines learn to use our strings in malware binaries as malware signatures. Adding these strings to a benign program will cause it to be detected as malware. Our findings show that the malware detection and labelling capabilities of VirusTotal engines are brittle and can be easily confused.

Bibliography

[1] D. Adams. *The Hitchhiker's Guide to the Galaxy*. San Val, 1995.

[2] Charu C. Aggarwal. Outlier ensembles [position paper], 2013.

[3] M. S. Akhtar, A. Ekbal, and E. Cambria. How intense are you? predicting intensities of emotions and sentiments using stacked ensemble [application notes]. *IEEE Computational Intelligence Magazine*, 15(1):64–75, 2020.

[4] Mohannad J. Alhanahnah, Qicheng Lin, Qiben Yan, N. Zhang, and Zhenxiang Chen. Efficient signature generation for classifying cross-architecture iot malware. *2018 IEEE Conference on Communications and Network Security (CNS)*, pages 1–9, 2018.

[5] Saed Alrabaee, Lingyu Wang, and Mourad Debbabi. Bingold: Towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (sfgs). *Digital Investigation*, 18:S11–S22, 08 2016.

[6] Omar Alrawi, Charles Lever, Kevin Valakuzhy, Ryan Court, Kevin Snow, Fabian Monrose, and Manos Antonakakis. The circle of life: A large-scale study of the iot malware lifecycle. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3505–3522. USENIX Association, August 2021.

[7] H. Anderson. Evading machine learning malware detection. 2017.

[8] 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.

[9] 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.

[10] 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.

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

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

[13] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the mirai botnet. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, 2017.

[14] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the mirai botnet. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1093–1110, Vancouver, BC, 2017. USENIX Association.

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

[16] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. BYTEWEIGHT: Learning to recognize functions in binary code. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 845–860, San Diego, CA, August 2014. USENIX Association.

[17] Erick Bauman, Zhiqiang Lin, and Kevin W. Hamlen. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. In *Proceedings 2018 Network and Distributed System Security Symposium*, San Diego, CA, 2018. Internet Society.

[18] Dima Ben, Igal. Breaking down mirai: An iot ddos botnet analysis. *https://www.imperva.com/blog/malware-analysis-mirai-ddos-botnet/*, 2016.

[19] M. Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. Speculative disassembly of binary code. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '16, New York, NY, USA, 2016. Association for Computing Machinery.

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

[21] Stevens Le Blond, Adina Uritesc, Cédric Gilbert, Zheng Leong Chua, Prateek Saxena, and Engin Kirda. A look at targeted attacks through the lense of an NGO. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 543–558, San Diego, CA, August 2014. USENIX Association.

[22] Gavin Brown, Jeremy Wyatt, Rachel Harris, and Xin Yao. Diversity creation methods: A survey and categorisation. *Journal of Information Fusion*, 6:5–20, 2005.

[23] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward Schwartz. Bap: A binary analysis platform. volume 6806, pages 463–469, 01 2011.

[24] Avast Carly Burdova. What is eternalblue and why is the ms17-010 exploit still relevant? *https://www.avast.com/c-eternalblue*, December 8, 2020.

[25] Jason Carolan. Digital devices took over our lives in 2020: Here's how to stay secure. *Online:https://www.forbes.com/sites/forbestechcouncil/2021/04/15/digital-devices-took-over-our-lives-in-2020-heres-how-to-stay-secure/?sh=5fd67df91ed2*, 2021.

[26] Mahinthan Chandramohan, Hee Beng Kuan Tan, and Lwin Khin Shar. Scalable malware clustering through coarse-grained behavior modeling. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, New York, NY, USA, 2012. Association for Computing Machinery.

[27] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. Bingo: Cross-architecture cross-os binary search. FSE 2016, page 678–689, New York, NY, USA, 2016. Association for Computing Machinery.

[28] Kai-Chi Chang, Raylin Tso, and Min-Chun Tsai. Iot sandbox: To analysis iot malware zollard. In *Proceedings of the Second International Conference on Internet of Things, Data and Cloud Computing*, ICC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[29] Kai Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. Finding unknown malice in 10 seconds: Mass vetting for new threats at the google-play scale. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 659–674, Washington, D.C., August 2015. USENIX Association.

[30] Lingwei Chen, Yanfang Ye, and Thirimachos Bourlai. Adversarial machine learning in malware detection: Arms race between evasion attack and defense. In *2017 European Intelligence and Security Informatics Conference (EISIC)*, pages 99–106, 2017.

[31] Mozammel Chowdhury, Azizur Rahman, and Rafiqul Islam. Malware analysis and detection using data mining and machine learning classification. In Jemal Abawajy, Kim-Kwang Raymond Choo, and Rafiqul Islam, editors, *International Conference on Applications and Techniques in Cyber Security and Intelligence*, pages 266–274, Cham, 2018. Springer International Publishing.

[32] Mihai Christodorescu and Somesh Jha. Testing malware detectors. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, page 34–44, New York, NY, USA, 2004. Association for Computing Machinery.

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

[34] Cisco. *Internet of Things - Cisco Report.* cisco, 2016.

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

[36] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 95–110, Berkeley, CA, USA, 2014. USENIX Association.

[37] 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.

[38] 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.

[39] Emanuele Cozzi, Pierre-Antoine Vervier, Matteo Dell'Amico, Yun Shen, Leyla Bilge, and Davide Balzarotti. The tangled genealogy of iot malware. In *Annual Computer Security Applications Conference*, ACSAC '20, page 1–16, New York, NY, USA, 2020. Association for Computing Machinery.

[40] Emanuele Cozzi, Pierre-Antoine Vervier, Matteo Dell'Amico, Yun Shen, Leyla Bilge, and Davide Balzarotti. The tangled genealogy of iot malware. In *Annual Computer Security Applications Conference*, ACSAC '20, page 1–16, New York, NY, USA, 2020. Association for Computing Machinery.

[41] Anusha Damodaran, Fabio Di Troia, Corrado Aaron Visaggio, Thomas H. Austin, and Mark Stamp. A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques*, 13(1):1–12, dec 2015.

[42] Berte Dan-Radu. Defining the IoT. *Proceedings of the International Conference on Business Excellence*, 12(1):118–128, May 2018.

[43] Hung Dang, Yue Huang, and Ee-Chien Chang. Evading classifiers by morphing in the dark. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 119–133, New York, NY, USA, 2017. Association for Computing Machinery.

[44] 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.

[45] Ahmad Darki and Michalis Faloutsos. Riotman: a systematic analysis of iot malware behavior. In *International Conference On Emerging Networking Experiments And Technologies (CoNEXT) pp. 169-182*. ACM, 2020.

[46] Yaniv David, Nimrod Partush, and Eran Yahav. Similarity of binaries through re-optimization. *SIGPLAN Not.*, 52(6):79–94, June 2017.

[47] Yaniv David, Nimrod Partush, and Eran Yahav. Similarity of binaries through re-optimization. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 79–94, New York, NY, USA, 2017. Association for Computing Machinery.

[48] Code de la Propriété Intellectuelle. Hopperv4. *https://www.hopperapp.com/*, 2012.

[49] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. Bistro: Binary component extraction and embedding for software security applications. In Crampton et al. [49], pages 200–218.

[50] V Diels-Grabsch, M Brand, T Theodore, M Gerhardy, T Timothy Gu, and B Nagae. Mxe (m cross environment). *https://mxe.cc/.*, 2007-2019.

[51] Saso Džeroski and Bernard Ženko. Is combining classifiers with stacking better than selecting the best one? *Mach. Learn.*, 54(3):255–273, March 2004.

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

[53] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovre: Efficient cross-architecture identification of bugs in binary code. 02 2016.

[54] Zhiyang Fang, Junfeng Wang, Boya Li, Siqi Wu, Yingjie Zhou, and Haiying Huang. Evading anti-malware engines with deep reinforcement learning. *IEEE Access*, 7:48867–48879, 2019.

[55] Alessandro Federico, Mathias Payer, and Giovanni Agosta. rev.ng: a unified binary analysis framework to recover cfgs and function boundaries. pages 131–141, 02 2017.

[56] Qian Feng, Aravind Prakash, Minghua Wang, Curtis Carmony, and Heng Yin. ORIGEN: automatic extraction of offset-revealing instructions for cross-version memory analysis. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, pages 11–22, 2016.

[57] 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.

[58] Sean Ford, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Analyzing and Detecting Malicious Flash Advertisements. In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC)*, Honolulu, HI, December 2009.

[59] Sri Shaila G, Ahmad Darki, Michalis Faloutsos, Nael Abu-Ghazaleh, and Manu Sridharan. Idapro for iot malware analysis? In *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*, Santa Clara, CA, August 2019. USENIX Association.

[60] Laurence Goasduff. Gartner says 5.8 billion enterprise and automotive iot endpoints will be in use in 2020. *Online: https://www.gartner.com/en/newsroom/press-releases/2019-08-29-gartner-says-5-8-billion-enterprise-and-automotive-io*, 2019.

[61] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial examples for malware detection. In Simon N. Foley, Dieter Gollmann, and Einar Snekkenes, editors, *Computer Security – ESORICS 2017*, pages 62–79, Cham, 2017. Springer International Publishing.

[62] Mahmoud Hammad, Joshua Garcia, and Sam Malek. A large-scale empirical study on the effects of code obfuscations on android apps and anti-malware products. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 421–431, New York, NY, USA, 2018. Association for Computing Machinery.

[63] Kyoung-Soo Han, Jae Hyun Lim, and Eul Gyu Im. Malware analysis method using visualization of binary files. In *RACS*, 2013.

[64] Irfan Ul Haq, Sergio Chica, Juan Caballero, and Somesh Jha. Malware lineage in the wild. *CoRR*, abs/1710.05202, 2017.

[65] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1667–1680, New York, NY, USA, 2018. ACM.

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

[67] Xin Hu, Kang G. Shin, Sandeep Bhatkar, and Kent Griffin. Mutantx-s: Scalable malware clustering based on static features. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 187–198, San Jose, CA, June 2013. USENIX Association.

[68] Médéric Hurier, Guillermo Suarez-Tangil, Santanu Kumar Dash, Tegawendé F. Bissyandé, Yves Le Traon, Jacques Klein, and Lorenzo Cavallaro. Euphony: Harmonious unification of cacophonous anti-virus vendor labels for android malware. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 425–435, 2017.

[69] Augusto Remillano II. Fbot aka satori is back with new peculiar obfuscation, brute-force techniques. *https://www.trendmicro.com/vinfo/za-en/security/news/internet-of-things/fbot-aka-satori-is-back-with-new-peculiar-obfuscation-brute-force-techniques*, 2019.

[70] Vector 35 Inc. Binary ninja. *https://docs.binary.ninja/index.html*, 2020.

[71] Kazuki Iwamoto and Katsumi Wasaki. Malware classification based on extracted api sequences using static analysis. In *Proceedings of the Asian Internet Engineeering Conference*, AINTEC '12, page 31–38, New York, NY, USA, 2012. Association for Computing Machinery.

[72] Muhui Jiang, Yajin Zhou, Xiapu Luo, Ruoyu Wang, Yang Liu, and Kui Ren. An empirical study on arm disassembly tools. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 401–414, New York, NY, USA, 2020. Association for Computing Machinery.

[73] Curtis Franklin Jr. 7 malware families ready to ruin your iot's day, 29 2019.

[74] Nikos Karampatziakis. Static analysis of binary executables using structural svms. In *NIPS*, 2010.

[75] Amin Kharraz, Sajjad Arshad, Collin Mulliner, William Robertson, and Engin Kirda. UNVEIL: A Large-Scale, Automated Approach to Detecting Ransomware. In *25th USENIX Security Symposium*, 08 2016.

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

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

[78] Johannes Kinder. Jakstab. *http://www.jakstab.org/*, 2010.

[79] Deguang Kong and Guanhua Yan. Discriminant malware distance learning on structural information for automated malware classification. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, page 1357–1365, New York, NY, USA, 2013. Association for Computing Machinery.

[80] Brian Krebs. Source code for iot botnet 'mirai' released. *Online:https://krebsonsecurity.com/2016/10/source-code-for-iot-botnet-mirai-released/*, 2016.

[81] Anders Krogh and Jesper Vedelsby. Neural network ensembles, cross validation and active learning. In *Proceedings of the 7th International Conference on Neural Information Processing Systems*, NIPS'94, page 231–238, Cambridge, MA, USA, 1994. MIT Press.

[82] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *13th USENIX Security Symposium (USENIX Security 04)*, San Diego, CA, August 2004. USENIX Association.

[83] C. Krügel, William K. Robertson, Fredrik Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *USENIX Security Symposium*, 2004.

[84] Nokia Threat Intelligence Lab. *Nokia Threat Intelligence Report – 2019*. Nokia, 2019.

[85] Evangelos Ladakis, Giorgos Vasiliadis, M. Polychronakis, S. Ioannidis, and G. Portokalidis. Gpu-disasm: A gpu-based x86 disassembler. In *ISC*, 2015.

[86] Bingchang Liu, Wei Huo, Chao Zhang, Wenchao Li, Feng Li, Aihua Piao, and Wei Zou. adiff: Cross-version binary code similarity detection with dnn. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 667–678, New York, NY, USA, 2018. ACM.

[87] Weibo Liu, Zidong Wang, Xiaohui Liu, Nianyin Zeng, Yurong Liu, and Fuad E. Alsaadi. A survey of deep neural network architectures and their applications. *Neurocomputing*, 234:11–26, 2017.

[88] Xinbo Liu, Jiliang Zhang, Yaping Lin, and He Li. Atmpa: Attacking machine learning-based malware visualization detection methods via adversarial examples. In *2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQoS)*, pages 1–10, 2019.

[89] Keane Lucas, Mahmood Sharif, Lujo Bauer, Michael K. Reiter, and Saurabh Shintre. Malware makeover: Breaking ml-based static analysis by modifying executable bytes. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '21, page 744–758, New York, NY, USA, 2021. Association for Computing Machinery.

[90] Keane Lucas, Mahmood Sharif, Lujo Bauer, Michael K. Reiter, and Saurabh Shintre. Malware makeover: Breaking ml-based static analysis by modifying executable bytes. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '21, page 744–758, New York, NY, USA, 2021. Association for Computing Machinery.

[91] Mcafee. What is petya and notpetya ransomware? *https://www.mcafee.com/enterprise/en-us/security-awareness/ransomware/petya.html*, 2021.

[92] Xiaozhu Meng. Dynist disassembler. *https://github.com/dyninst/dyninst/releases/tag/v10.1.0*, 2016.

[93] Xiaozhu Meng and Barton P. Miller. Binary code is not easy. In *ISSTA 2016*, 2016.

[94] Merrium-Webster. percentile. *https://www.merriam-webster.com/dictionary/percentile*, 2020.

[95] Trend Micro. Source code of iot botnet satori publicly released on pastebin. *Online:https://www.trendmicro.com/vinfo/za-en/security/news/internet-of-things/source-code-of-iot-botnet-satori-publicly-released-on-pastebin*, 2018.

[96] K. Miller, Y. Kwon, Y. Sun, Z. Zhang, X. Zhang, and Z. Lin. Probabilistic disassembly. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1187–1198, 2019.

[97] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. Probabilistic disassembly. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 1187–1198. IEEE Press, 2019.

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

[99] Aziz Mohaisen and Omar Alrawi. Av-meter: An evaluation of antivirus scans and labels. In Sven Dietrich, editor, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 112–131, Cham, 2014. Springer International Publishing.

[100] F. Mohsen, E. Bello-Ogunu, and M. Shehab. Investigating the keylogging threat in android — user perspective (regular research paper). In *2016 Second International Conference on Mobile and Secure Services (MobiSecServ)*, pages 1–5, Feb 2016.

[101] M. Nar, A. G. Kakisim, M. N. Yavuz, and İ. Soğukpinar. Analysis and comparison of disassemblers for opcode based malware analysis. In *2019 4th International Conference on Computer Science and Engineering (UBMK)*, pages 17–22, Sep. 2019.

[102] Lakshmanan Nataraj, Vinod Yegneswaran, Phillip Porras, and Jian Zhang. A comparative assessment of malware classification using binary texture analysis and dynamic analysis. 09 2011.

[103] netspooky. threatlands. *https://github.com/threatland/TL-BOTS*, 2019.

[104] Peter Newman. Iot report: How internet of things technology growth is reaching mainstream companies and consumers. *https://www.businessinsider.com/internet-of-things-report?IR=T.*, 2019.

[105] Quoc-Dung Ngo, Huy-Trung Nguyen, Van-Hoang Le, and Doan-Hieu Nguyen. A survey of iot malware and detection methods based on static features. *ICT Express*, 6(4):280–286, 2020.

[106] Norton. The future of iot: 10 predictions about the internet of things, 2016.

[107] Lina Nouh, Ashkan Rahimian, Djedjiga Mouheb, Mourad Debbabi, and Aiman Hanna. Binsign: Fingerprinting binary functions to support automated analysis of code executables. In Sabrina De Capitani di Vimercati and Fabio Martinelli, editors, *ICT Systems Security and Privacy Protection*, pages 341–355, Cham, 2017. Springer International Publishing.

[108] NSA. Ghidraorg. *https://ghidra-sre.org/*, 2019.

[109] Krebs on Security. Study: Attack on krebsonsecurity cost iot device owners $323k. *Online:https://krebsonsecurity.com/2018/05/study-attack-on-krebsonsecurity-cost-iot-device-owners-323k/*, May 7, 2018.

[110] GCC GNU ORG. Options that control optimization. *https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html*, 2020.

[111] Nickname: pancake and. Radare2. *https://github.com/radareorg*, 2020.

[112] 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.

[113] James Patrick-Evans, Lorenzo Cavallaro, and Johannes Kinder. Probabilistic naming of functions in stripped binaries. In *Annual Computer Security Applications Conference*, ACSAC '20, page 373–385, New York, NY, USA, 2020. Association for Computing Machinery.

[114] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy*, pages 709–724, May 2015.

[115] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-architecture bug search in binary executables. In *2015 IEEE Symposium on Security and Privacy*, pages 709–724, May 2015.

[116] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 709–724, Washington, DC, USA, 2015. IEEE Computer Society.

[117] R. Qiao and R. Sekar. Function interface analysis: A principled approach for function recognition in cots binaries. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 201–212, 2017.

[118] Rui Qiao and Ramachandran Sekar. Function interface analysis: A principled approach for function recognition in cots binaries. 06 2017.

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

[120] Lior Rokach. Ensemble-based classifiers. *Artif. Intell. Rev.*, 33(1–2):1–39, February 2010.

[121] Md Omar Faruk Rokon, Risul Islam, Ahmad Darki, Evangelos E. Papalexakis, and Michalis Faloutsos. Sourcefinder: Finding malware source-code from publicly available

repositories in github. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 149–163, San Sebastian, October 2020. USENIX Association.

[122] Nathan Rosenblum, Xiaojin Zhu, Barton Miller, and Karen Hunt. Learning to analyze binary computer code. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2*, AAAI'08, page 798–804. AAAI Press, 2008.

[123] Unit 42 Ruchna Nigam. Unit 42 finds new mirai and gafgyt iot/linux botnet campaigns. *https://unit42.paloaltonetworks.com/unit42-finds-new-mirai-gafgyt-iotlinux-botnet-campaigns/*, 2016.

[124] Akamai Ryan Barnett, Principal Security Researcher. New tsunami/kaiten variant: Propagation status. *https://blogs.akamai.com/sitr/2018/09/new-tsunamikaiten-variant-propagation-status.html*, September 11, 2018.

[125] Omer Sagi and L. Rokach. Ensemble learning: A survey. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8, 2018.

[126] S. Sagiroglu and G. Canbek. Keyloggers: Increasing threats to computer security and privacy. *IEEE Technology and Society Magazine*, 28(3):10–17, Fall 2009.

[127] O. P. Samantray, S. N. Tripathy, and S. K. Das. A study to understand malware behavior through malware analysis. In *2019 IEEE International Conference on System, Computation, Automation and Networking (ICSCAN)*, pages 1–5, 2019.

[128] Robert E. Schapire, Yoav Freund, Peter Barlett, and Wee Sun Lee. Boosting the margin: A new explanation for the effectiveness of voting methods. In *Proceedings of the Fourteenth International Conference on Machine Learning*, ICML '97, page 322–330, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[129] National Security Agency Central Security Service. National security agency central security service. *https://www.nsa.gov/*, 2020.

[130] Sri Shaila, Ahmad Darki, Michalis Faloutsos, Nael Abu-Ghazaleh, and Manu Sridharan. Disco: Combining disassemblers for improved performance. In *24th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '21, page 148–161, New York, NY, USA, 2021. Association for Computing Machinery.

[131] P.V. Shijo and A. Salim. Integrated static and dynamic analysis for malware detection. *Procedia Computer Science*, 46:804–811, 12 2015.

[132] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing functions in binaries with neural networks. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 611–626, Washington, D.C., August 2015. USENIX Association.

[133] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, 2016.

[134] Rami Sihwail, K. Omar, and K. A. Z. Ariffin. A survey on malware analysis techniques: Static, dynamic, hybrid and memory analysis. *International Journal on Advanced Science, Engineering and Information Technology*, 8:1662–1671, 2018.

[135] H. Sinanović and S. Mrdovic. Analysis of mirai malicious software. In *2017 25th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pages 1–5, 2017.

[136] Jagsir Singh and Jaswinder Singh. A survey on machine learning-based malware detection in executable files. *J. Syst. Archit.*, 112:101861, 2021.

[137] Avast Software. What is eternalblue and why is the ms17-010 exploit still relevant? *https://www.avast.com/c-eternalblue(December8, 2020*, 2020.

[138] Avast Software. Mcafee. 2021. what is petya and not-petya ransomware? *https://www.mcafee.com/enterprise/en-us/securityawareness/ransomware/petya.html*, 2021.

[139] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: Having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, page 1808–1815, New York, NY, USA, 2013. Association for Computing Machinery.

[140] NSA Developers Sri Shaila G. Reduced function start identification in ghidra9.1 compared to ghidra 9.0.4 for arm binaries. *https://github.com/NationalSecurityAgency/ghidra/issues/1532*, Feb 2020.

[141] Gianluca Stringhini, Oliver Hohlfeld, Christopher Kruegel, and Giovanni Vigna. The harvester, the botmaster, and the spammer: On the relations between the different actors in the spam landscape. ASIA CCS '14, page 353–364, New York, NY, USA, 2014. Association for Computing Machinery.

[142] J. Su, D. V. Vasconcellos, S. Prasad, D. Sgandurra, Y. Feng, and K. Sakurai. Lightweight classification of iot malware based on image recognition. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 02, pages 664–669, 2018.

[143] Rui Tanabe, Tatsuya Tamai, Akira Fujita, Ryoichi Isawa, Katsunari Yoshioka, Tsutomu Matsumoto, Carlos Gañán, and Michel van Eeten. Disposable botnets: Examining the anatomy of iot botnet infrastructure. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, ARES '20, New York, NY, USA, 2020. Association for Computing Machinery.

[144] Gilbert Tanner. A guide to ensemble learning increase your accuracy by combining model outputs, 2019.

[145] Kurt Thomas, Elie Bursztein, Chris Grier, Grant Ho, Nav Jagpal, Alexandros Kapravelos, Damon Mccoy, Antonio Nappa, Vern Paxson, Paul Pearce, Niels Provos, and

Moheeb Abu Rajab. Ad injection at scale: Assessing deceptive advertisement modifications. In *2015 IEEE Symposium on Security and Privacy*, pages 151–167, 2015.

[146] Sadegh Torabi, Mirabelle Dib, Elias Bou-Harb, Chadi Assi, and Mourad Debbabi. A strings-based similarity analysis approach for characterizing iot malware and inferring their underlying relationships. *IEEE Networking Letters*, 3(3):161–165, 2021.

[147] Virus Total. Virustotal-free online virus, malware and url scanner. *Online:https://support.virustotal.com/hc/en-us/articles/115002126889-How-it-works*, 2012.

[148] Virus Total. Contributors. *Online:https://support.virustotal.com/hc/en-us/articles/115002146809-Contributors*, 2021.

[149] Center for Long-Term Cybersecurity UC Berkeley School of Information. Quantifying iot insecurity costs. *Online:https://groups.ischool.berkeley.edu/riot/*, 2018.

[150] UCSB. angr. *https://docs.angr.io/*, 2016.

[151] Iowa State University. Control flow analysis, 2016.

[152] US-CERT. US Computer Emergency Readiness Team heightened ddos threat posed by mirai and other botnets, alert ta16-288a. Accessed: 2016-11-30.

[153] Verizon. 2015databrenchinvestigationsreport, 29 2015.

[154] 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.

[155] VT. Virustotal. *https://www.virustotal.com/gui/*, 2020.

[156] Wikipedia contributors. Basic block — Wikipedia, the free encyclopedia, 2019. [Online; accessed 19-May-2019].

[157] Nicky Woolf. Ddos attack that disrupted internet was largest of its kind in history, experts say. *https://www.theguardian.com/technology/2016/oct/26/ddos-attack-dyn-mirai-botnet*, 2016.

[158] 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.

[159] Cha Zhang and Yunqian Ma. *Ensemble Machine Learning: Methods and Applications*. Springer Publishing Company, Incorporated, 2012.

[160] Mingwei Zhang and R. Sekar. Control flow integrity for cots binaries. SEC'13, page 337–352, USA, 2013. USENIX Association.

[161] Zhi-Hua Zhou. *Ensemble Methods: Foundations and Algorithms*. Chapman and Hall-CRC, 1st edition, 2012.

[162] Shuofei Zhu, Jianjun Shi, Limin Yang, Boqin Qin, Ziyi Zhang, Linhai Song, and Gang Wang. Measuring and modeling the label dynamics of online anti-malware engines. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2361–2378. USENIX Association, August 2020.

[163] Shuofei Zhu, Jianjun Shi, Limin Yang, Boqin Qin, Ziyi Zhang, Linhai Song, and Gang Wang. Measuring and modeling the label dynamics of online anti-malware engines. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2361–2378. USENIX Association, August 2020.