DisCo: Combining Disassemblers for Improved Performance

Sri Shaila G
University of California, Riverside
USA
sg001@ucr.edu

Ahmad Darki
University of California, Riverside
USA
adark001@ucr.edu

Michalis Faloutsos
University of California, Riverside
USA
michalis@cs.ucr.edu

Nael Abu-Ghazaleh
University of California, Riverside
USA
nael@cs.ucr.edu

Manu Sridharan
University of California, Riverside
USA
manu@cs.ucr.edu

ABSTRACT

Malware infects thousands of systems globally each day causing millions of dollars in damages. 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.

CCS CONCEPTS

• **Software and its engineering** → **Software organization and properties**; • **Computing methodologies** → *Ensemble methods*; • **Security and privacy** → **Software and application security**.

RAID '21, October 6–8, 2021, San Sebastian, Spain
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9058-3/21/10.
https://doi.org/10.1145/3471621.3471851

KEYWORDS

Ensemble, Disassembly Tools, ARM and MIPS architecture, Ghidra Bug Discovery, Improving Disassembly

ACM Reference Format:
Sri Shaila G, Ahmad Darki, Michalis Faloutsos, Nael Abu-Ghazaleh, and Manu Sridharan. 2021. *DisCo*: Combining Disassemblers for Improved Performance . In *24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '21), October 6–8, 2021, San Sebastian, Spain.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3471621.3471851

1 INTRODUCTION

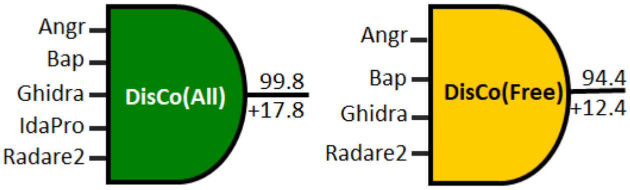

Figure 1: *DisCo* effectively combines disassemblers for superior performance: The gain in the performance over the best input can be as high as +17.8% with a combined performance 99.8% in F1 score combining five disassemblers. The results shown are for malware binaries compiled with GCC for MIPS with 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%).

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 [11, 32]. 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 [4, 56]. 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 [12, 18]. Note that determining the compilation parameters for a given stripped binary is a challenging task. Despite some recent studies [2, 25], 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 [25] evaluates several disassemblers using only benign binaries and for the ARM architecture only. Another work [2] evaluates disassemblers for the x86-64 architecture. Both studies [2, 25] 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 6.

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 side-stepping 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 [25]. 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 improvement, RPI. In table 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 1. Considering only our four non-commercial disassemblers, from table 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 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.

2 BACKGROUND

In this section, we present the background concepts and discuss the dataset that we use in this study.

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

Disassembler performance metrics. We use the **correctly identified function starts (*CFS*)**, which is the percentage of function start addresses that a disassembler detects correctly, as a primary metric. In general, instruction and function start identification are considered as the two fundamental metrics to evaluate disassemblers because they produce the output of other metrics like control-flow and call graphs [25]. Since previous work on ARM binaries found that disassemblers perform poorly for *CFS* [25], we opted to focus, investigate and improve on this metric. We believe that our approach of combining capabilities of multiple disassemblers applies to other metrics as well.

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 [34]. 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.

The data set. 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 [43]. 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 [5, 51–53, 56]. 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. [15]. Furthermore, security researchers have noticed new variants belonging to these malware families that are used to launch thousands of attacks in recent years [45, 46]. Source codes of over hundred variants of malware belonging to these families have been traced back to online repositories [15]. 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 4.1. We also discuss the coverage of the data set in section 5.

As shown in figure 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.

The five baseline disassemblers: We consider 5 disassemblers, Angr [55], IDA Pro [23], Ghidra [38], BAP [10], and Radare2 [39] in our work. IDA Pro performed the best in previous studies and other disassemblers have been used in recent evaluation studies [2, 25]. 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 [15]. 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 [21]. Furthermore, around half of all ELF malware are stripped [15].

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 [15]. 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.

3 OVERVIEW OF *DISCO*

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 disassemblers will only be beneficial if each disassembler recovers different parts of the binary structure. Individual disassemblers can miss some function starts (false negatives) 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.

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

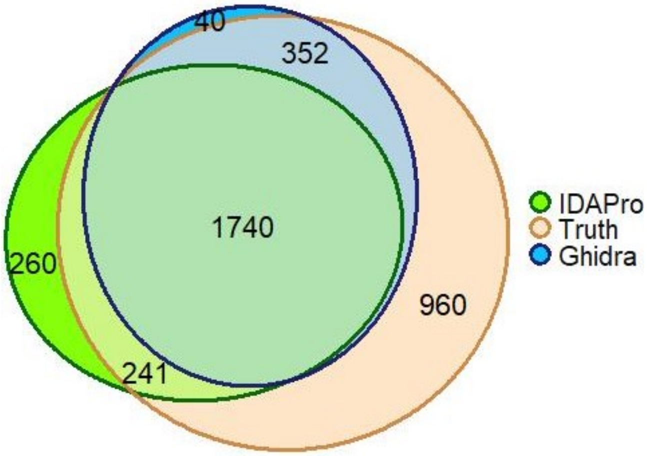

Figure 2: Motivating observation: Disassemblers complement each other. IDA Pro identifies 241 function starts and Ghidra 352 that the other does not identify. Similarly, the falsely identified function starts (260 and 40) seem to be disjoint. An efficient combination could improve the overall performance.

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.

3.1 The Architecture

Figure 3 shows the high level architecture of *DisCo*. 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.

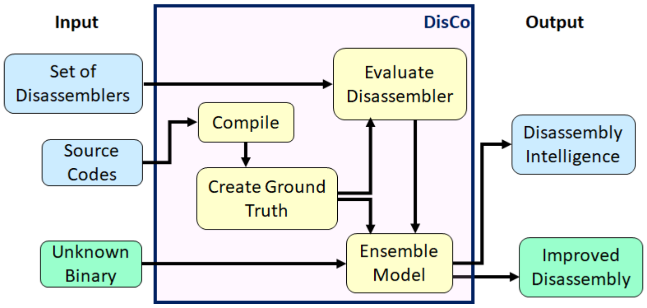

Figure 3: An overview of *DisCo* and its functional modules.

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 3.

c. Creating the ensemble model: An ensemble model gives superior prediction performance by integrating multiple models effectively [42]. 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 [54]. 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 [1, 37], 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 function 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.

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 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.

3.2 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.

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 [25]. 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[25]. 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 3. Note that it is well established that Radare2 does not support ARM binaries compiled with Clang [25].

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.

Table 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.

	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 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.

	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

4 EXPERIMENTAL EVALUATION

We evaluate the effectiveness of our approach with the datasets which we described in section 2 using the ground-truth which we discussed in section 3.2. 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.

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 1. In tables, 2 and 3. we show the performance of *DisCo* variants, *Ghidra+* variants and other disassemblers for *CFS* metric. Table 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 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 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 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 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 3, we see even larger improvement for the 5PWC values between *DisCo(All)* and the other two *DisCo* variants.

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 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.

Table 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.

	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

We observe a similar phenomenon in table 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. We noticed that disassembler performance varies greatly even for binaries of the same configuration. In table 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 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 3, 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 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 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 2 and 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.

Observation 5: Our systematic evaluation discovers a bug in Ghidra. In the creating of the training data for our ensemble model, we evaluate each disassembler. 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 4 , 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. 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

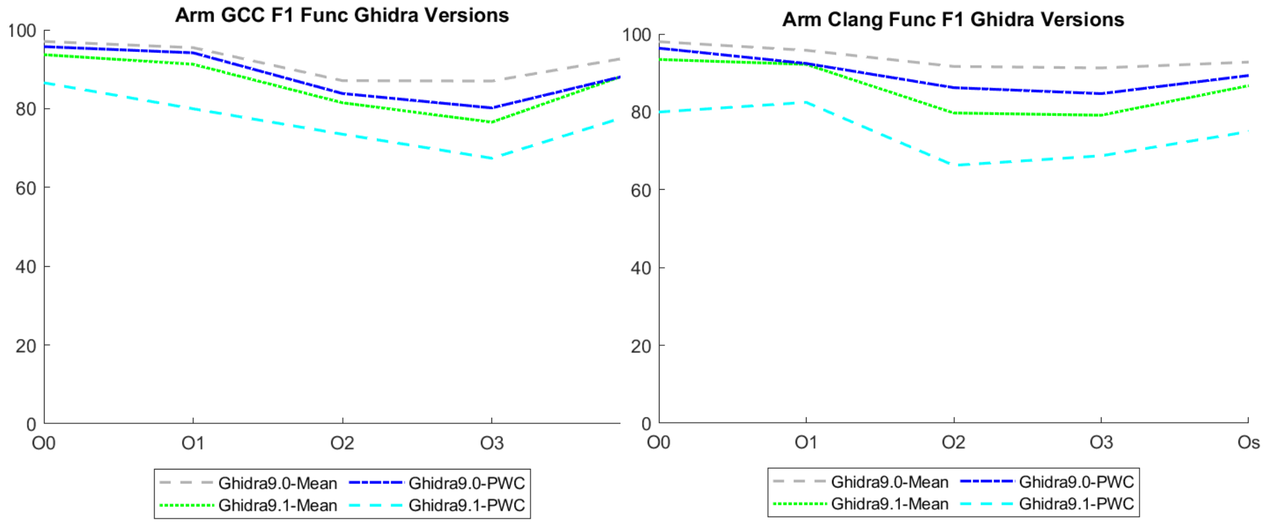

Figure 4: Ghidra v9.0.4 outperforms Ghidra v9.1: Mean F1 Score for ARM binaries compiled with GCC and Clang.

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[3] 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.

Q3: Which factors affect disassemble performance? Quantifying how different factors affect the performance of disassemblers is a challenging question. A related question is to identify which disassembler should be used for in scenario. Note that in practice, given an unknown binary, we do not necessarily know which configuration case produced it.

Observation 5: 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 2 and 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 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 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 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 5 and 6 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 8 and 9 in section **??**. 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 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.

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

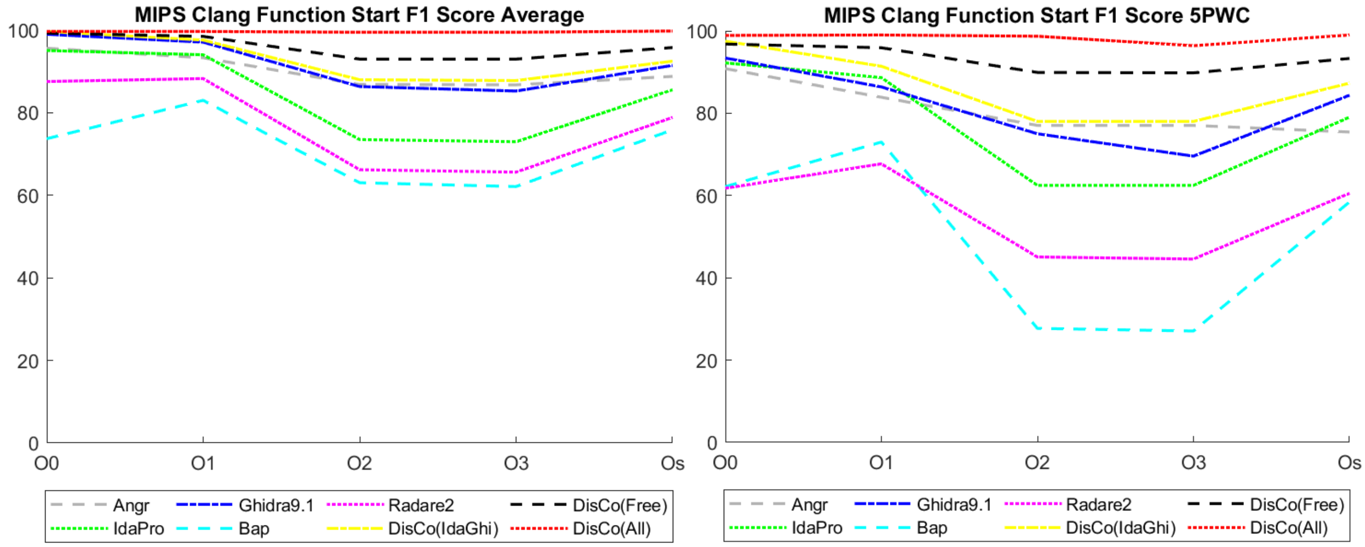

Figure 5: Average and 5PWC for F1 Score for Function Start Identification for MIPS binaries compiled with Clang.

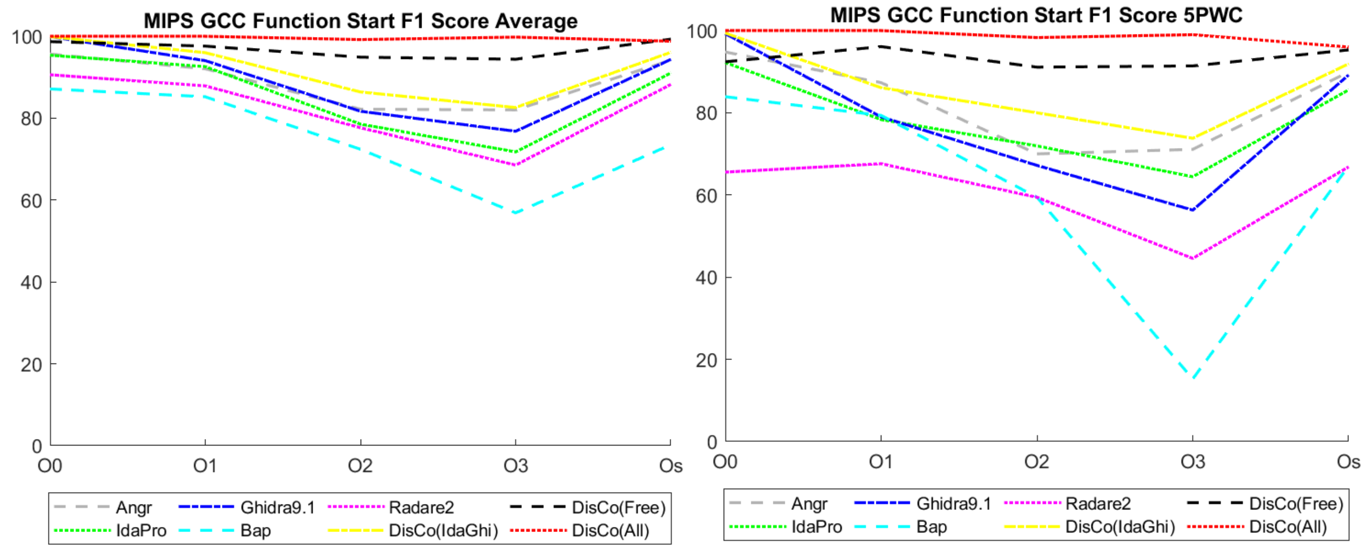

Figure 6: Average and 5PWC for F1 Score for Function Start Identification for MIPS binaries compiled with GCC.

4.1 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 7 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.

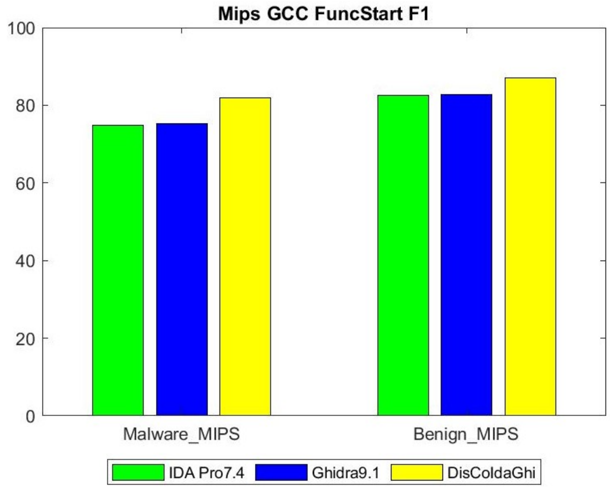

Figure 7: Benign and malware binaries: *DisCo* improves the performance even in the case of benign binaries. *DisCo* improves the F1 score by 6.7% for malware and 4.4% for benign binaries. Disassembling benign binaries seems easier. The reported results are for the MIPS architecture with GCC and the O3 compilation level.

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.

5 DISCUSSION

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 [19] and Binary Ninja [24], to further improve the combined performance. We can share our preliminary results with Binary Ninja [24]. 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 [25]. 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

Table 4: Time requirements for various disassemblers: We show the average time required for each disassembler for each binary.

	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

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 2. 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 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 4.1, 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[2, 3, 8]. 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[25].

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 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.

6 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 [25] 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 [2] 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 [21] 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 [3] 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 [29, 31, 57]. Other works present techniques like superset disassembly, probabilistic disassembly, and static analysis based method for x86 binaries [7, 35, 41]. Some works propose machine learning techniques for disassemebly [26] and to identify function starts [6, 10, 44, 49]. However, later works found that some of these works suffer from evaluation bias [3]. Other approaches use heuristics and or well-known function signatures to identify function starts [30, 50, 55]. Another work [20] 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 [40] 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 [36] 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[8], 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. [6, 10, 19, 23, 24, 27, 33, 33, 38, 55].

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 [14, 16, 17, 22] that create platforms that manage to activate IoT malware malware. Another work [13] 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 [47] 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 [9, 28]. Another work finds the reason behind trends in test errors in voting methods [48].

7 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.

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.

REFERENCES

[1] M. S. Akhtar, A. Ekbal, and E. Cambria. 2020. How Intense Are You? Predicting Intensities of Emotions and Sentiments using Stacked Ensemble [Application Notes]. *IEEE Computational Intelligence Magazine* 15, 1 (2020), 64–75. https://doi.org/10.1109/MCI.2019.2954667

[2] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. 2016. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 583–600.

[3] D. Andriesse, A. Slowinska, and H. Bos. 2017. Compiler-Agnostic Function Detection in Binaries. In *2017 IEEE European Symposium on Security and Privacy (EuroS P)*. 177–189. https://doi.org/10.1109/EuroSP.2017.11

[4] Anna-senpai. 2016. [FREE] world's largest net:Mirai botnet, client, echo loader, CNC source code release. *https://hackforums.net/showthread.php?tid=5420472.* (2016).

[5] 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. 2017. Understanding the Mirai Botnet. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association.

[6] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 845–860. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/bao

[7] Erick Bauman, Zhiqiang Lin, and Kevin W. Hamlen. 2018. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. In *Proceedings 2018 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA. https://doi.org/10.14722/ndss.2018.23300

[8] M. Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. 2016. Speculative Disassembly of Binary Code. In *Proceedings of the International Conference on*

Compilers, Architectures and Synthesis for Embedded Systems (Pittsburgh, Pennsylvania) *(CASES '16)*. Association for Computing Machinery, New York, NY, USA, Article 16, 10 pages. https://doi.org/10.1145/2968455.2968505

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

[10] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward Schwartz. 2011. BAP: A binary analysis platform. *LNCS* 6806, 463–469. https://doi.org/10.1007/978-3-642-22110-1_37

[11] Avast Carly Burdova. December 8, 2020. What Is EternalBlue and Why Is the MS17-010 Exploit Still Relevant? *https://www.avast.com/c-eternalblue* (December 8, 2020).

[12] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. BinGo: Cross-Architecture Cross-OS Binary Search *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 678–689. https://doi.org/10.1145/2950290.2950350

[13] Kai-Chi Chang, Raylin Tso, and Min-Chun Tsai. 2017. IoT Sandbox: To Analysis IoT Malware Zollard. In *Proceedings of the Second International Conference on Internet of Things, Data and Cloud Computing* (Cambridge, United Kingdom) *(ICC '17)*. Association for Computing Machinery, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3018896.3018898

[14] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. 2014. A Large-Scale Analysis of the Security of Embedded Firmwares.. In *USENIX Security Symposium*.

[15] Emanuele Cozzi, Pierre-Antoine Vervier, Matteo Dell'Amico, Yun Shen, Leyla Bilge, and Davide Balzarotti. 2020. The Tangled Genealogy of IoT Malware. In *Annual Computer Security Applications Conference* (Austin, USA) *(ACSAC '20)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/3427228.3427256

[16] Ahmad Darki, Chun-Yu Chuang, Michalis Faloutsos, Zhiyun Qian, and Heng Yin. 2018. RARE: A Systematic Augmented Router Emulation for Malware Analysis. In *International Conference on Passive and Active Network Measurement*. Springer, 60–72.

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

[18] Yaniv David, Nimrod Partush, and Eran Yahav. 2017. Similarity of Binaries through Re-Optimization. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 79–94. https://doi.org/10.1145/3062341.3062387

[19] Code de la Propriété Intellectuelle. 2012. HopperV4. *https://www.hopperapp.com/* (2012).

[20] Alessandro Federico, Mathias Payer, and Giovanni Agosta. 2017. rev.ng: a unified binary analysis framework to recover CFGs and function boundaries. 131–141. https://doi.org/10.1145/3033019.3033028

[21] Sri Shaila G, Ahmad Darki, Michalis Faloutsos, Nael Abu-Ghazaleh, and Manu Sridharan. 2019. IDAPro for IoT Malware analysis?. In *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/cset19/presentation/g

[22] Claudio Guarnieri, Allessandro Tanasi, Jurriaan Bremer, and Mark Schloesser. 2012. The cuckoo sandbox.

[23] SA Hex-Rays. 2008. IDA pro disassembler.

[24] Vector 35 Inc. 2020. Binary Ninja. *https://docs.binary.ninja/index.html* (2020).

[25] Muhui Jiang, Yajin Zhou, Xiapu Luo, Ruoyu Wang, Yang Liu, and Kui Ren. 2020. An Empirical Study on ARM Disassembly Tools. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 401–414. https://doi.org/10.1145/3395363.3397377

[26] Nikos Karampatziakis. 2010. Static Analysis of Binary Executables Using Structural SVMs. In *NIPS*.

[27] Johannes Kinder. 2010. Jakstab. *http://www.jakstab.org/* (2010).

[28] Anders Krogh and Jesper Vedelsby. 1994. Neural Network Ensembles, Cross Validation and Active Learning. In *Proceedings of the 7th International Conference on Neural Information Processing Systems* (Denver, Colorado) *(NIPS'94)*. MIT Press, Cambridge, MA, USA, 231–238.

[29] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. 2004. Static Disassembly of Obfuscated Binaries. In *13th USENIX Security Symposium (USENIX Security 04)*. USENIX Association, San Diego, CA. https://www.usenix.org/conference/13th-usenix-security-symposium/static-disassembly-obfuscated-binaries

[30] C. Krügel, William K. Robertson, Fredrik Valeur, and G. Vigna. 2004. Static Disassembly of Obfuscated Binaries. In *USENIX Security Symposium*.

[31] Evangelos Ladakis, Giorgos Vasiliadis, M. Polychronakis, S. Ioannidis, and G. Portokalidis. 2015. GPU-Disasm: A GPU-Based X86 Disassembler. In *ISC*.

[32] Mcafee. 2021. What Is Petya and NotPetya Ransomware? *https://www.mcafee.com/enterprise/en-us/security-awareness/ransomware/petya.html* (2021).

[33] Xiaozhu Meng. 2016. Dynist Disassembler. *https://github.com/dyninst/dyninst/releases/tag/v10.1.0* (2016).

[34] Merrium-Webster. 2020. percentile. *https://www.merriam-webster.com/dictionary/percentile* (2020).

[35] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. 2019. Probabilistic Disassembly. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 1187–1198. https://doi.org/10.1109/ICSE.2019.00121

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

[37] Quoc-Dung Ngo, Huy-Trung Nguyen, Van-Hoang Le, and Doan-Hieu Nguyen. 2020. A survey of IoT malware and detection methods based on static features. *ICT Express* 6, 4 (2020), 280–286. https://doi.org/10.1016/j.icte.2020.04.005

[38] NSA. 2019. GhidraOrg. *https://ghidra-sre.org/* (2019).

[39] Nickname: pancake and. 2020. Radare2. *https://github.com/radareorg* (2020).

[40] James Patrick-Evans, Lorenzo Cavallaro, and Johannes Kinder. 2020. Probabilistic Naming of Functions in Stripped Binaries. In *Annual Computer Security Applications Conference* (Austin, USA) *(ACSAC '20)*. Association for Computing Machinery, New York, NY, USA, 373–385. https://doi.org/10.1145/3427228.3427265

[41] R. Qiao and R. Sekar. 2017. 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)*. 201–212. https://doi.org/10.1109/DSN.2017.29

[42] Lior Rokach. 2010. Ensemble-Based Classifiers. *Artif. Intell. Rev.* 33, 1–2 (Feb. 2010), 1–39. https://doi.org/10.1007/s10462-009-9124-7

[43] Md Omar Faruk Rokon, Risul Islam, Ahmad Darki, Evangelos E. Papalexakis, and Michalis Faloutsos. 2020. SourceFinder: Finding Malware Source-Code from Publicly Available Repositories in GitHub. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. USENIX Association, San Sebastian, 149–163. https://www.usenix.org/conference/raid2020/presentation/omar

[44] Nathan Rosenblum, Xiaojin Zhu, Barton Miller, and Karen Hunt. 2008. Learning to Analyze Binary Computer Code. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2* (Chicago, Illinois) *(AAAI'08)*. AAAI Press, 798–804.

[45] Unit 42 Ruchna Nigam. 2016. Unit 42 Finds New Mirai and Gafgyt IoT/Linux Botnet Campaigns. *https://unit42.paloaltonetworks.com/unit42-finds-new-mirai-gafgyt-iotlinux-botnet-campaigns/* (2016).

[46] Akamai Ryan Barnett, Principal Security Researcher. September 11, 2018. NEW TSUNAMI/KAITEN VARIANT: PROPAGATION STATUS. *https://blogs.akamai.com/sitr/2018/09/new-tsunamikaiten-variant-propagation-status.html* (September 11, 2018).

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

[48] Robert E. Schapire, Yoav Freund, Peter Barlett, and Wee Sun Lee. 1997. Boosting the Margin: A New Explanation for the Effectiveness of Voting Methods. In *Proceedings of the Fourteenth International Conference on Machine Learning (ICML '97)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 322–330.

[49] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing Functions in Binaries with Neural Networks. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 611–626. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/shin

[50] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. 138–157. https://doi.org/10.1109/SP.2016.17

[51] H. Sinanović and S. Mrdovic. 2017. Analysis of Mirai malicious software. In *2017 25th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*. 1–5. https://doi.org/10.23919/SOFTCOM.2017.8115504

[52] J. Su, D. V. Vasconcellos, S. Prasad, D. Sgandurra, Y. Feng, and K. Sakurai. 2018. Lightweight Classification of IoT Malware Based on Image Recognition. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 02. 664–669. https://doi.org/10.1109/COMPSAC.2018.10315

[53] Rui Tanabe, Tatsuya Tamai, Akira Fujita, Ryoichi Isawa, Katsunari Yoshioka, Tsutomu Matsumoto, Carlos Gañán, and Michel van Eeten. 2020. Disposable Botnets: Examining the Anatomy of IoT Botnet Infrastructure. In *Proceedings of the 15th International Conference on Availability, Reliability and Security* (Virtual Event, Ireland) *(ARES '20)*. Association for Computing Machinery, New York, NY, USA, Article 7, 10 pages. https://doi.org/10.1145/3407023.3409177

[54] Gilbert Tanner. 2019. A guide to Ensemble Learning Increase your accuracy by combining model outputs.

[55] UCSB. 2016. angr. *https://docs.angr.io/* (2016).

[56] US-CERT. [n.d.]. US Computer Emergency Readiness Team Heightened DDoS Threat Posed by Mirai and Other Botnets, alert TA16-288A. Accessed: 2016-11-30.

[57] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries *(SEC'13)*. USENIX Association, USA, 337–352.

8 APPENDIX

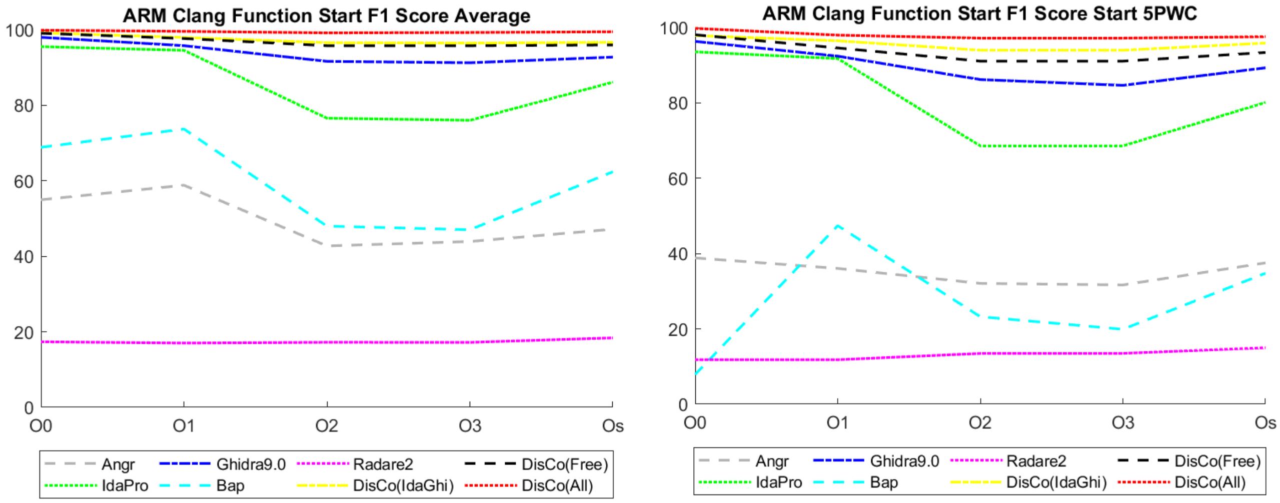

Figure 8: Average and 5PWC for F1 Score for Function Start Identification for ARM binaries compiled with Clang.

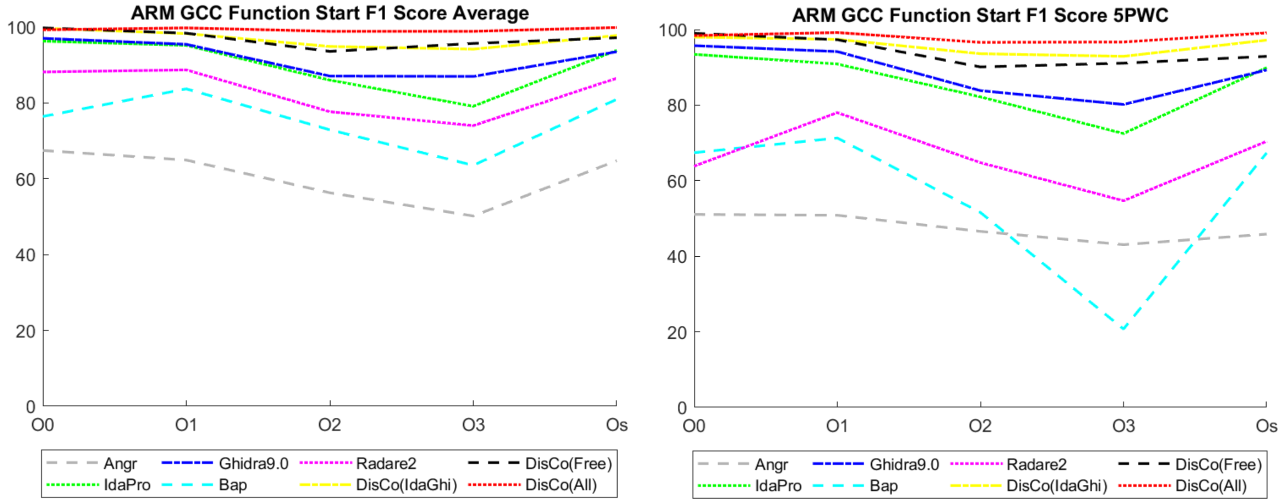

Figure 9: Average and 5PWC for F1 Score for Function Start Identification for ARM binaries compiled with GCC.