MalFusion: Simple string manipulations confuse malware detection

Sri Shaila G[1] and Michalis Faloutsos[2]

[1]Palo Alto Networks, Email: gsrishaila@gmail.com
[2]University of California, Riverside, Email: michalis@cs.ucr.edu

***Abstract*—We explore how a hacker confuse malware detection engines with minimal effort. As our overarching contribution, we show that such engines can be easily manipulated to misclassify malware and benign binaries. We substantiate our claim by developing *MalFusion*, a framework to stress-test and confuse anti-malware engines in identifying malware and its family type. The design goal for our method is to be as simple as possible and without changing the binary functionality. We apply *MalFusion* on 1750 binaries compiled with different compilers and compiler options for ARM and MIPS architectures and we use 71 anti-malware engines provided by VirusTotal. The overarching observation is that the anti-malware engines rely heavily on source-level string matching such as strings in printf commands. First, we show that when one of our simple string manipulation techniques is applied to malware source-code, 100% of the binaries are deemed as benign. Second, we observe that engines learn: within two weeks, they identify our modified binaries as malware. Third, we show how to exploit this "learning" capability by making engines misclassify string-modified benign binaries. Finally, we observe that there is no free lunch: engines with higher recall on malware binaries are prone to false positives.**

***Index Terms*—virtual evasion, malware code manipulation, antivirus evasion malware classification, assessment, measurement, practice/deployment**

I. Introduction

With a projected 41 billion IoT devices worldwide by 2025 [1], the opportunity for hackers is huge. The recent Mirai attack in 2016 shows the urgent need to secure 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 [2] 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. Furthermore, studies [3] find an increasing number of IoT malware repositories containing source codes since 2015. .All trends show an increased activity for IoT malware [3], [4].

Taking a hacker-centric view, we explore approaches to confuse malware detection engines. In more detail, we frame the question with the following constraints and assumptions. First, putting ourselves in the position of the hacker, we assume that we have *access to the malware source code*. Second, we require any obfuscation technique to have the following characteristics: (a) require minimal effort to apply, and (b) preserve the program's functionality. We opt to focus on IoT malware,

ISBN 978-3-903176-63-8 ©2024 IFIP

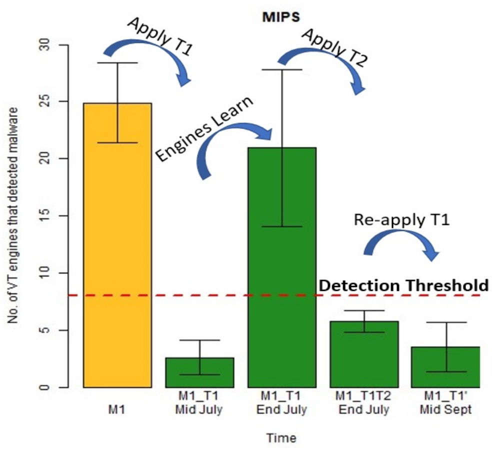

Fig. 1: Our simple techniques can confuse anti-malware engines effectively and repeatedly. In our MIPS malware dataset: (a) applying technique T1 reduces the detection rate from roughly 25 to 3 engines, (b) the engines "learn" to recognize the submitted binaries going back to 21 engines within two weeks, (c) applying technique T2 reduces detection to 6 engines, and (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 the number of engines required to detect malware confidently.

which is known to have unique characteristics [5], [6] and require novel solutions [7]. We focus on the MIPS and ARM architectures, because they are widely used in IoT devices [8]. Although the problem and our methods can apply to other architectures, we do not investigate this here.

There is relatively limited work on the problem as we frame it here. Unlike most previous work, we consider and focus on: (a) the learning capabilities of the engines, (b) modifying the source code, (c) the misclassification of benign programs, and (d) IoT malware. Specifically, we can group previous efforts in the following categories. The first category focuses on modifying the malware *binaries* to evade detection [9]–[11]. Another category uses binaries found in the wild to evaluate the detection capabilities of VirusTotal engines [12], [13] without proposing evasion techniques. Some other approaches modify the control flow graph to evade detection [14], [15], which we consider too "intrusive" of a change. Finally, some much earlier studies focus on modifying source-code, but focused on Windows machines and Visual Basic [16]. We discuss related work in detail in section VI.

In this work, we conduct an extensive study to show that *painfully* simple string manipulations at the source code level can confuse malware detection and classification. We develop *MalFusion* (**Mal**ware Detection Con**fusion**), a framework that includes a set of simple string-manipulation techniques and we systematically check the responses that anti-malware engines give. Our manipulation techniques fit our problem requirements, as they: (a) require minimal effort, and (b) do not alter the program functionality. In fact, we argue that its the simplicity of these techniques that makes our work interesting and somewhat surprising. First, we show that these engines rely heavily on string matching. Thus, modifying strings in the source-code (think strings in a printf command) can lead to misclassifications. Second, we are arguably the first study that observes and quantifies that these engines learn: our submitted binaries are used by the detection capability within two weeks.

We conduct a study with 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 (**VT**). We study two functionalities of the engines: (a) detecting malware, and (b) identifying the malware family. Our key observations are captured in figure 1.

a. Engines rely heavily on source-code level strings. We develop two painfully simple yet effective techniques causing 100% and 95% of malware to be declared as benign respectively. **Technique T1** manipulates strings in the source code (e.g. “Connection established”) by adding an arbitrary two-character string between every pair of characters in the initial strings. **Technique T2** adds a “dummy” line of code between every two lines in the source code.

Applying T1 leads to a decrease in detection rate from 22 to around 2 engines on average. Figure 1 showcases this drop in the detection rate, and also captures the additional steps in our study, which we discuss below.

b. Engines learn new “malware” strings within two weeks. We observe that within at most two weeks, the engines learn to identify the obfuscated malware, and we have roughly 20 engines that classify the same modified binaries as malware. Applying T2 after makes this number of engines go down to roughly 5. T1 technique can also be reused with a new input string to evade detection.

c. Learning can be exploited: arbitrary strings become malware signatures. We show how hackers can exploit the combination of: (a) the learning capability, and (b) the heavy reliance on strings. This can be done with the following steps. First, we add an arbitrary string in a malware source code and submit the binary to VirusTotal. Second, we wait for two weeks for the engines to “learn”. Third, we add the string to a benign program, and the binary is now classified as malware. In our experiments, even “high-reputation“ engines like Avast, AVG and Fortinet are fooled. Furthermore, this misclassification carries over in the identification of the malware family. For example, adding Mirai strings into benign code, we get 89% of the binaries to be considered Mirai, 6% surprisingly labeled as Gafgyt, and only 5% identified as benign.

d. 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 ($\geq$84.9% in our malware datasets) are also more likely to misclassify modified benign binaries.

Note that we fully reported our experiments to security engineers at VirusTotal and the anti-malware engines.

II. Background and Datasets

We discuss related background and our datasets.

A. Background and definitions. 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 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 engine e. Note that we can check consistency at two levels of granularity: (a) the detection of malware, and (b) classification to a malware family.

Note that we do not consider precision because there are no false positives in our datasets. This is because no sample can be falsely classified as malware in the malware datasets and no sample can be falsely classified as benign in the benign dataset.

The malware classification threshold (T). Previous works suggests ways to combine the verdicts of 71 engines in VirusTotal. For example, a binary can be considered to be malware if 2-15 engines flag it as malware [13] with 8.5 being in the middle of this range. Here, we set the threshold, T to 8. Recent work also suggests that this threshold can show the maliciousness of a sample stably over time [17]. Therefore, **we consider a binary to be malware when 8 or more engines flag it as malware**. Note that varying this threshold will mostly affect the trade off between false negatives and false positives.

B. Binary variations: architecture and compilation. Binaries in our dataset were compiled with the following compilation configurations. We did this because a malware author can also employ these techniques. This is because creating multiple binaries, which are functionally equivalent, but vary at the binary code level, can maximize the chance of evading detection. This can be easily achieved by using different compilation configurations [18].

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 [19].

Dataset	Description
M1	First set of malware binaries
M1_T1	Modified with technique T1 with string “PD”
M1_T1T2	Modified with technique T1 and T2
M1_T1’	Modified with T1 with string ”*∧”
M2	Second set of malware binaries
M2_T1	Modified with technique T1
B1	Initial set of benign binaries
B1_MS	Expanded with malware strings
B1_OMS	Expanded with our modified malware strings

TABLE I: The binary datasets used in our study, malware M1 and M2, and benign B1 and their modified versions. MS refers to the strings found in the malware source code, while OMS refers to the malware strings after applying T1.

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 [20]. 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: (a) instruction substitution, (b) bogus control flow, and (c) 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 because around half of all ELF malware are stripped and they are challenging for malware analysis tools [19], [21].

C. Our datasets: We have used a total of 1750 binaries in our study across different architectures, compilation options, and variations. Specifically, we have two malware datasets with distinct binaries: the **M1 dataset** with 280 malware binaries, and the **M2 dataset** with 240 malware binaries. Both datasets contain binaries that were compiled with all the compilation configurations described above. The code and dataset will be made available on request. Table I lists the key datasets in our study.

We explain how we created our datasets and their variations. 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. All the programs that we have used in our study have been used in recent studies on malware [21], [22]. We use 7 of these programs to create the M1 dataset and the rest to create the M2 dataset. We explain our choice of datasets in Section V.

We also created a small main benign binary dataset, B1 containing 10 binaries. We obtained them by compiling a well-known benign program, 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 C language.

Creating modified malware datasets. To evaluate the engines, we apply our techniques on our datasets, which we describe below. 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 manipulation technique(s) that was applied on the source code to obtain the binaries in that dataset.

Our definition of source-code level strings. We define source-code level strings to be all characters found between a pair of unescaped double quotes in the source code. These strings often are: (a) user messages, such as “Failed to open socket”, and (b) malware communication protocol messages, such as “Bot deploy failed” [8], [23].

a. Modifying malware source code. We create variations of dataset M1 by applying: (a) manipulation technique T1 on M1 to create M1_T1 with string parameter “PD”, (b) technique T2 on M1_T1 to get M1_T1T2, (c) reapplying T1 on M1 with string parameter ”*∧” to get M1_T1’. We also applied similar variations to the M2 dataset.

b. Modifying benign source code. We generated B1_MS and B1_OMS in the following way. We extracted strings from 7 *initial malware programs* (refer to as MS) from M1 and added each of these to each B1 program to form B1_MS with 70 binaries. We extracted *our modified strings* from 7 of the modified programs (referred to as OMS) from M1_T1 and added them to each B1 binaries to form B1_OMS with 70 binaries. MS stands for malware strings and OMS stands for our manipulated strings. We preserved the program functionality by adding the strings into a string array and by using a for loop to ”use” the strings in the array. We refer to the binaries in B1_MS as string-added binaries, while we refer to B1_OMS binaries as string-modified binaries.

D. Scalable assessment of engines. An efficient way to evaluate the majority of prominent anti-malware engines is through the widely-used VirusTotal platform [24]. VirusTotal is a free online service that analyzes binaries by using 71 anti-malware engines. VirusTotal reports how each engine classifies a binary: (a) malicious or benign, and (b) malware family [24], [25]. VirusTotal has been extensively used in research to label datasets and evaluate tools [13], [25]–[32]. We are grateful to the VirusTotal team as their work makes our research possible.

III. Overview of our approach

We present key ideas and insights in developing the *MalFusion* framework. Its purpose is to provide techniques to confuse anti-malware, with the constraints that they are: (a) easy to apply, and (b) preserve the original functionality.

The simple ideas that should not have worked. We present the insights that led to two simple methods that we thought would never work.

a. Manipulating source code strings. First, we thought of focusing on source-code level strings. As we defined earlier, these strings consist of: (a) protocol communication, and (b) program notification messages. For example, this definition includes strings that appear in the communication between IoT malware and its C&C server [8]. Upon investigation, we found that the source code of IoT programs contain many such strings. We wanted to check if these strings would influence the detection of malware.

b. Introducing dummy commands. We wanted to check if the sequence of commands in identifying malware. 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 commands in the source code that will not change the way the malware behaves.

Initially, we thought that the engines would consider the semantics of a binary code and conduct sophisticated analysis of the structure of the binary. Since our techniques preserves the malware behaviour, we thought that they were way too simple to be effective.

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

a. 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. Though the choice of the string parameter can be important, it seems that a two-character parameter works well in our study.

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 randomly chosen 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 this function with "PD" as its parameter and T1' to refer to cases where we use this function with "*∧".

Ensuring no functional change. Note that we can 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 each pair of original characters. 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 remain 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 generated.

b. Technique T2: In this technique, we add a non-functional statement between each two lines of code in the source code. In our study, we use a *printf* command with a string. However, other commands which do not cause any functional change can be used as well. 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.

Assessing temporal consistency and adaptation. We also repeat our evaluation to assess: (a) the consistency of engines over time, and (b) their ability to learn and adapt. For example, we repeat an evaluation with the exact same binaries at a later time. We also create different binaries using the same technique but using different input string in technique T1.

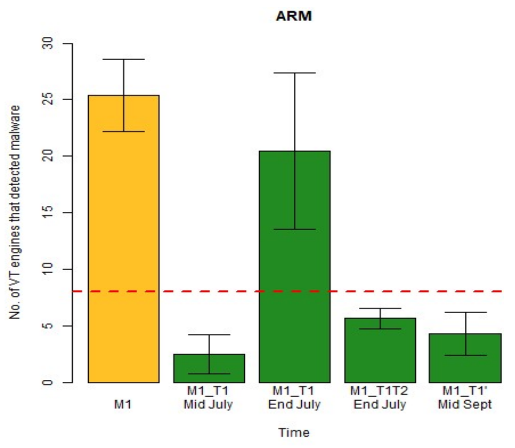

Fig. 2: **Our simple techniques can evade VirusTotal engines repeatedly and effectively for 100% of the ARM binaries in M1 dataset:** Applying T1 on ARM binaries in M1 reduces the average number of engines that detect the malware from 25.4 to 2.49. Two weeks later, applying T2 on the binaries reduces average number of engines that detect malware from 20.5 to 5.7. Applying T1' two months after we applied T1 reduces the average number of engines that detect malware from 25.4 to 4.3.

IV. Experimental Evaluation

We test the robustness of the engines by applying our techniques on the datasets described in section II. Due to space limitations, we only show a subset of the results. We will provide more results in the journal version. We group our experimental results around the following three questions, which we answer with a series of observations.

Q1: How effective are our simple evasion techniques?

Q2: Can we make benign programs appear as malware?

Q3: Are some engines more trustworthy?

Summary of the resuls: Engines rely heavily on source-code level strings for malware detection and family classification. For most engines, modifying source-code level strings leads to evasion causing false negatives. Adding strings found in malware binaries to benign binaries makes the latter be detected as malware, leading to false positives. Finally, injecting modified strings also affects the malware family classification.

Q1: How effective are our simple evasion techniques?

Our results show that both our manipulation 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 1 and 2. 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 manipulation 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 1 and 2 respectively. Note that we repeat the first two steps with dataset M2, which corroborated our observations.

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. The average number of engines that flagged binaries as malware in M1 is between 25 and 26.5 for both architectures when Clang4 or Clang4(OLLVM) is used. 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.

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 1 and 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. 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 respectively. The standard deviation for both architectures is 6.9. This observation is an indication that engines are updated frequently which is aligned with previous studies [13].

Unfortunately, as we will see in later observations, at least a part of this learning is only "string-deep", which means that it would be 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 1 and 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 by using new string parameters. We apply technique T1 with a new string parameter "*∧" to create dataset M1_T1' and 100% of its binaries evade detection. As shown in Figures 1 and 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 manipulation techniques. A decrease in the number of engines that detected a malware binary as malware indicates that our manipulation techniques have modified some features in the binary that are being used by engines to detect malware. Observations 1 - 4 suggest

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 II: 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.

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. We keep repeating this step until we have the minimum number of lines of code that is needed to be identified by the engines as malware. We find that a small set of source-code strings is enough to confuse the engines. We use this technique to make observations 6 and 8 below.

Observation 6: Engines misclassify benign programs purely based on source-level strings. We add the strings found in each malware program in M1 separately to a simple bubble sort program described in Section II 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 “use” the strings in the array in the main function of the program. We did this to prevent the situation where unused variables get omitted during the compilation process. We compiled all the modified sorting programs 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.

The benign binaries are now considered to be malware. Introducing these strings 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 is comparable to the number of engines that detected the corresponding malicious binaries in M1 (25.8 for ARM and 26.7 MIPS). As a result, all binaries in our benign dataset B1_MS are classified as malware, as they have significantly more than 8 engines vouching against them in October 2021.

Interestingly, the misclassification persists a year later. Resubmitting the B1_MS samples in October 2022 showed the average number of misclassifying engines increased from 22.9 to 35.9 and 23.3 to 35.7 respectively. The standard deviation for the modified sorted programs is 2.8 and 2.0 for the ARM and MIPS binaries respectively.

Few strings can manipulate the verdict. 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 misclassifications. In Table II, we show sets of strings that were sufficient to confuse the engines. The introduction of these strings in a simple sorting program is sufficient to misclassify it as malware. Furthermore, some engines give different results for the modified benign binary for different architectures. For example, Kaspersky considers the ARM version of the benign binary as malware, but not the MIPS version when ”GET /fuck1hex” string is added.

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 of strings shown in Table II and a for loop to print them. This short program of around 10 lines make the engines in the table flag them as malware. This suggests that at least some engines classify a program based solely on blacklisted strings.

Observation 7: Engines rely on strings for malware family classification. We go deeper into the misclassification of string-modified benign binaries and examine what malware family is reported. We find that the malware family of 83.1% and 79.3% of the string-modified 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.

In more detail, we focus on three major IoT malware families: Gafgyt, Mirai and Tsunami [19]. 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 [12], [33] does not affect our observations here.

We introduce strings from a malware family into benign code and observe what is the family label that the engines identify. By introducing Tsunami strings for ARM binaries, all the benign binaries were reported as Tsunami. In this case, although wrongly reported as malware, at least it is reported as the correct family. By introducing Mirai strings for MIPS binaries, we find that 89% of the binaries are labeled as Mirai, 6% as Gafgyt, and 5% as benign. These results were indicative of the confusion that the introduction of malware-derived strings can cause.

Observation 8: We can force arbitrarily created strings to become malware signatures. We take our ability to manipulate the anti-malware engines further by establishing our arbitrarily created strings as malware signatures. Recall that we create B1_OMS by inserting strings that we added in malware binaries in benign code. We find that a substantial number of engines detect binaries in B1_OMS as malware. The average number of engines that detect the ARM and the MIPS binaries in

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 III: Search engines rely on string matching: the strings that we have created eventually become "signatures" for malware: adding these strings in benign software makes some engines flag them as malware.

B1_OMS is 6.9 and 7.2 with a standard deviation is 0.9 and 1.1 respectively, which brings the result close to the threshold of 8 engines. In fact, one of the ARM binaries and five of the MIPS binaries exceeded the threshold of 8 engines, which means that they would have been reported as malware.

Over time the misclassification becomes worse. When we resubmitted the samples in October 2022, we noticed that average number of engines that detected the ARM and MIPS modified benign binaries increased from 6.9 to 22.3 and 7.3 to 23.4 respectively. This shows that the misclassification not only did not get corrected, but on the contrary, it got worse.

Similar to our findings in observation 6, even a small subset of strings from M1_T1 is sufficient to make benign binaries be misclassified as malware. We show indicatively sets of strings from M1_T1 that were sufficient for misclassification by the engines listed next to them in Table III. As reference, we also provide the strings before our modification. Binaries containing these strings were detected as malware by the same engines for both architectures. Interestingly, the presence of the corresponding initial versions of these strings (MS) in benign binaries do not cause misclassification. For example, the presence of the initial string "(null)" in a benign binary does not trigger the Fortinet engine, but the modified version of the string "(PDnPDuPDlPDlPD)" makes Fortinet to misclassify it as malware.

This observation shows that malware authors can influence the set of string level signatures that are used by the engines. This provides opportunities for a new type of "attacks" on the credibility of these engines as we discuss in section V.

Q3: Are some engines more trustworthy?

So far, we have examined the capability of engines as a group. Here, we investigate which engines are more reliable for the different architectures and compilation options. We quantify the reliability of an engine by using recall and their consistency as defined earlier.

The best performing engines that give the top 10 scores for recall and consistency are: Avast-Mobile, AVG, Avast, Fortinet, TrendMicro-HouseCall, Ikarus, GData, Microsoft MicroWorld-eScan, BitDefender, MAX, Ad-Aware, McAfee-GW-Edition, McAfee, FireEye and Emsisoft. 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 average consistency of the engines as follows. We focus on three versions of corresponding malware binaries, M1, M1_T1, M1_T1T2. For each engine, we calculate the consistency as the average consistency between M1 and M1_T1, and M1 and M1_T1T2. 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 for each engine. We then average the two consistency values for each engine. The top engines give an average consistency score of 84.2% or higher for ARM binaries and 84.7% or higher for MIPS binaries.

Observation 9: No-free-lunch: The higher-recall engines on malware datasets exhibit higher misclassifation for the modified benign datasets, B1_MS and B1_OMS. We find that all these top-performing engines misclassified at least 20% of the binaries in the B1_MS dataset. The list of engines that misclassify at least 80% of the binaries in B1_MS and B1_OMS for ARM and MIPS also contain all but four of these top-performing engines. The better performing four engines, TrendMicro-HouseCall, Ikarus, McAfee and McAfee-GW-Edition misclassified 20-75% of the binaries in B1_MS.

Observation 10: Technique T1 affects malware family 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 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.

Out of curiosity, we decided to investigate the percentage of pairs of binaries from M1 and M1_T1 that were classified as belonging to the same major family and same sub-classification or version by the engines. We observed 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 family labelling functionality in engines relies heavily on strings.

Observation 11: String manipulation affects recall rates of even for some high-reputation engines. Recent studies have collectively and informally suggested a group of "high-reputation" engines[1] [13], [34], [35]. However, even these engines are affected by our simple

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

manipulation techniques, often, by a significant drop in their detection: from 70-99% down to 0! In Table IV, we show the results for some of these high-reputation engines. Recall that, in M1_T1T2, we use both our techniques: (T1) string modification and (T2) insertion of commands. Kaspersky gives almost perfect recall for binaries in M1 for both architecture, but gives 0% recall for binaries in M1_T1 and M1_T1T2. ESET-NOD32 has at least 97% recall for binaries in M1, and drops only to 72-77% with technique T1, but drops to zero when both techniques are applied. Sophos has a 70-75% recall for binaries in M1 but gives 0% recall for the corresponding binaries in M1_T1 and M1_T1T2. F-Secure consistently gives a recall of below 10% for both architectures for all four malware datasets, Finally, Symantec exhibits a peculiar behavior: it starts with 29.3-30% detection for M1 which increases to 61.4-64.3% for M1_T1 and drops to 29.3% for M1_T1T2. We leave this as future work.

	ARM			MIPS		
Engines	**M1**	**M1_ T1**	**M1_ T1T2**	**M1**	**M1_ T1**	**M1_ T1T2**
Kaspersky	99.3	0	0	100	0	0
ESET-NOD32	99.3	72.1	0	97.1	77.1	0
Sophos	75.0	0	0	70.7	0	0
F-Secure	2.1	4.3	5.7	3.6	7.1	9.3
Symantec	29.3	61.4	29.3	30	64.3	29.3

TABLE IV: Some high-reputation engines exhibit poor recall on modified malware.

Observation 12: The choice of compiler affects malware detection rates. We find that the top 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% for MIPS. The recall rates given by the other three compilers for this dataset for both architectures is above 90%.

V. Discussion

We discuss the broader context and limitations.

a. Practical implications of our approach. Analysts and engineers can use our approach to stress-test the effectiveness of anti-malware engines. Specifically, our approach can test if an engine is able to: (a) detect both the initial and the modified versions of a malware binary, (b) detect malware binaries that are compiled with various compilers and compilation options, (c) avoid false positives, and (d) can correctly identify the malware family label consistently across modified binaries and different compilation configurations.

b. Our dataset comprises of major malware families. 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, we are using real malware, since: (a) it is offered as malware by their GitHub authors, and (b) the combined wisdom of the engines classifies it as such in its initial version.

c. Comparing various versions of VirusTotal: Previous work compared the detection capability of the desktop version to the version in VirusTotal [13]. However, the VirusTotal version of the malware is widely used from the research community [13], [25]–[32].

d. Implications and misbehavior opportunities. At the highest level, our study shows that a systematic offensive from hackers can potentially render anti-malware engines in VirusTotal ineffective. First, hackers can create undetectable malware by simply modifying strings and adding dummy lines of code, as shown in Observations 1 and 3. Second, a systematic campaign by hackers could add so much "noise" in the malware signature database that could render the results untrustworthy. Observations 6 and 8 show that the anti-malware engines continue to misclassify benign binaries even after one year after we started our study. We speculated that engines would have processes in place to sanitized the database periodically, but such processes are absent.

e. Mitigation: Engines can address this vulnerability by using harder-to-fake features or incorporate reverse engineering steps to reveal the actual or check if the binary contains information about any C2S server found in open-access database of C2S server before assigning botnet related labels to these binaries [36]. Note that recent literature has proposed techniques that use commands or control flow graphs that could provide more reliable results [5], [15]. We intend to investigate mitigation techniques in future.

f. Limitation and future work: Further experiments are needed to find out if our techniques are effective on other malware families. We leave this and the potential mitigation strategies as future work.

VI. Related Work

To the best of our knowledge, there has not been any previous study that focuses on applying simple manipulation 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) engines evasion via source code manipulation, and (c) engines evaluation over a period of one year. We group relevant research in the following categories.

a. Modifying source-code to evade detection: Two recent works [14], [15] modify the control flow graph to stress test their own classification technique, not VirusTotal. By contrast, we do not inject code snippets: we do minimal and non-functional modifications.

b. Modifying binaries to evade detection: A large body of work focus on modifying malware *binaries* to avoid detection. A recent work [25] discusses string manipulations on IoT malware in a white-box setting which is not applicable for VirusTotal engines. It also evaluates VirusTotal engines by using manipulated IoT binaries created by using packing, stripping and padding techniques. However, the majority of these binaries were detected by more that 8 engines. The majority of prior studies focused on Windows platforms in contrast we focus on IoT platforms [9]–[11], [37]–[41].

d. Conducting longitudinal and robustness studies: Some recent works assess the stability of the classification provided by VirusTotal engines and study the classification effectiveness for over a year [13], [17], [42] but without doing any active modifications of the binaries, as we do here. Other works evaluate the engines at only one point in time [12], [43] and attempt to identify the more reliable among all the engines [44], [45]. None of these studies proposes evasion methods, as we do here.

VII. Conclusion

In this work, we also explore the limits of the capabilities of anti-malware engines by pretending to be malware authors. We propose *MalFusion*, a framework to stress-test and confuse VirusTotal engines.

a. Avoiding detection with simple tricks. We find that our two simple techniques, string manipulations and adding "filler" lines of code, are sufficient to give high evasion rates of above 95%. These solutions preserve the original functionality and require minimal effort.

b. Poisoning the malware signature database. We find that engines place significant weight on source code level strings and hackers can exploit this "capability" by adding them to benign programs.

We believe that our study is a significant step in understanding how security engines detect and label malware. We identify and document several weaknesses in the performance of these engines during the span of more than a year. We envision that our framework will be used to stress-test and improve the capabilities of VirusTotal engines.

This work was supported by NSF SaTC No. 2132642.

References

[1] J. Carolan, "Digital devices took over our lives in 2020: Here's how to stay secure," *Forbes Magazine*, April 11 2021.
[2] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou, "Understanding the mirai botnet," in *USENIX Security 2017*.
[3] M. O. F. Rokon, R. Islam, A. Darki, E. E. Papalexakis, and M. Faloutsos, "SourceFinder: Finding Malware Source-Code from Publicly Available Repositories in GitHub," in *RAID*, 2020.
[4] R. 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.
[5] H. Alasmary, A. Khormali, A. Anwar, J. Park, J. Choi, A. Abusnaina, A. Awad, D. Nyang, and A. Mohaisen, "Analyzing and detecting emerging internet of things malware: A graph-based approach," *IEEE Internet of Things Journal*, 2019.
[6] A. Anwar, "Statically dissecting internet of things malware: Analysis, characterization, and detection," *Lecture notes in computer science*, 2020.
[7] A. Darki and M. Faloutsos, "RIoTMAN: A systematic analysis of IoT malware behavior," in *CoNEXT*, 2020.
[8] O. Alrawi, C. Lever, K. Valakuzhy, R. Court, K. Snow, F. Monrose, and M. Antonakakis, "The Circle Of Life: A Large-Scale Study of The IoT Malware Lifecycle," in *USENIX Security 2021*.
[9] K. Lucas, M. Sharif, L. Bauer, M. K. Reiter, and S. Shintre, "Malware makeover: Breaking ml-based static analysis by modifying executable bytes," in *ASIA CCS 2021*.
[10] Z. Fang, J. Wang, B. Li, S. Wu, Y. Zhou, and H. Huang, "Evading anti-malware engines with deep reinforcement learning," *IEEE Access*, 2019.
[11] H. S. Anderson, A. Kharkar, B. Filar, and P. Roth, "Evading machine learning malware detection," *BlackHat*, 2017.
[12] A. Mohaisen and O. Alrawi, "Av-meter: An evaluation of antivirus scans and labels," in *DIMVA 2014*.
[13] S. Zhu, J. Shi, L. Yang, B. Qin, Z. Zhang, L. Song, and G. Wang, "Measuring and modeling the label dynamics of online anti-malware engines," in *USENIX Security 2020*.
[14] A. Abusnaina, H. Alasmary, M. Abuhamad, S. Salem, D. Nyang, and A. Mohaisen, "Subgraph-based adversarial examples against graph-based IoT malware detection systems," in *CSoNet 2019*.
[15] A. Abusnaina, A. Khormali, H. Alasmary, J. Park, A. Anwar, and A. Mohaisen, "Adversarial learning attacks on graph-based IoT malware detection systems," in *ICDCS 2019*.
[16] M. Christodorescu and S. Jha, "Testing malware detectors," in *ISSTA 2004*.
[17] J. Wang, L. Wang, F. Dong, and H. Wang, "Re-measuring the label dynamics of online anti-malware engines from millions of samples," in *Proceedings of the 2023 ACM on Internet Measurement Conference*, ser. IMC '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 253–267. [Online]. Available: https://doi.org/10.1145/3618257.3624800
[18] Y. David, N. Partush, and E. Yahav, "Similarity of binaries through re-optimization," in *PLDI 2017*.
[19] E. Cozzi, P.-A. Vervier, M. Dell'Amico, Y. Shen, L. Bilge, and D. Balzarotti, "The tangled genealogy of IoT malware," in *ACSAC 2020*.
[20] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-llvm – software protection for the masses," in *SPRO 2015*.
[21] S. S. G, A. Darki, M. Faloutsos, N. Abu-Ghazaleh, and M. Sridharan, "IDAPro for IoT Malware analysis?" in *CSET 2019*.
[22] S. Shaila, A. Darki, M. Faloutsos, N. Abu-Ghazaleh, and M. Sridharan, "DisCo: Combining Disassemblers for Improved Performance," in *RAID 2021*.
[23] D. D. Jovanovic and P. V. Vuletic, "Analysis and characterization of IoT malware command and control communication," in *TELFOR 2019*.
[24] "Virustotal - free online virus, malware and url scanner," 2021.
[25] A. Abusnaina, A. Anwar, S. Alshamrani, A. Alabduljabbar, R. Jang, D. Nyang, and D. Mohaisen, "Systematically evaluating the robustness of ml-based IoT malware detection systems," in *RAID 2022*.
[26] K. Chen, P. Wang, Y. Lee, X. Wang, N. Zhang, H. Huang, W. Zou, and P. Liu, "Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale," in *USENIX Security 2015*.
[27] S. Ford, M. Cova, C. Kruegel, and G. Vigna, "Analyzing and detecting malicious flash advertisements," in *ACSAC 2009*.
[28] M. Hammad, J. Garcia, and S. Malek, "A large-scale empirical study on the effects of code obfuscations on android apps and anti-malware products," in *ICSE 2018*.
[29] A. Kharraz, S. Arshad, C. Mulliner, W. Robertson, and E. Kirda, "Unveil: A large-scale, automated approach to detecting ransomware," in *USENIX Security 2016*.
[30] S. L. Blond, A. Uritesc, C. Gilbert, Z. L. Chua, P. Saxena, and E. Kirda, "A look at targeted attacks through the lense of an ngo," in *USENIX Security 2014*.
[31] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann, "Mobile-sandbox: Having a deeper look into android applications," in *SAC '13*.
[32] G. Stringhini, O. Hohlfeld, C. Kruegel, and G. Vigna, "The harvester, the botmaster, and the spammer: On the relations between the different actors in the spam landscape," in *ASIA CCS '14*.
[33] M. Hurier, G. Suarez-Tangil, S. K. Dash, T. F. Bissyandé, Y. Le Traon, J. Klein, and L. Cavallaro, "Euphony: Harmonious unification of cacophonous anti-virus vendor labels for android malware," in *MSR 2017*.
[34] M. Chandramohan, H. B. K. Tan, and L. K. Shar, "Scalable malware clustering through coarse-grained behavior modeling," in *FSE 2012*.
[35] D. Kong and G. Yan, "Discriminant malware distance learning on structural information for automated malware classification," in *KDD 2013*.
[36] V. Jain, S. Alam, S. Krishnamurthy, and M. Faloutsos, "C2store: C2 server profiles at your fingertips," *Proceedings of the ACM on Networking*, vol. 1, pp. 1–21, 11 2023.
[37] X. Liu, J. Zhang, Y. Lin, and H. Li, "Atmpa: Attacking machine learning-based malware visualization detection methods via adversarial examples," in *IWQoS 2019*.
[38] H. Dang, Y. Huang, and E.-C. Chang, "Evading classifiers by morphing in the dark," in *CCS 2017*.
[39] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, "Adversarial examples for malware detection," in *ESORICS 2017*.
[40] J. Yuste, E. G. Pardo, and J. Tapiador, "Optimization of code caves in malware binaries to evade machine learning detectors," in *Elsevier 2022*.
[41] B. Chen, Z. Ren, C. Yu, I. Hussain, and J. Liu, "Adversarial examples for cnn-based malware detectors," *IEEE Access*, 2019.
[42] S. Zhu, Z. Zhang, L. Yang, L. Song, and G. Wang, "Benchmarking label dynamics of virustotal engines," in *CCS 2020*.
[43] D. Cocca, A. Pirozzi, and C. A. Visaggio, "We cannot trust in you: A study about the dissonance among anti-malware engines." in *ARES 2022*.
[44] E. Raff, B. Filar, and J. Holt, "Getting passive aggressive about false positives: Patching deployed malware detectors," in *ICDMW 2020*.
[45] M. Botacin, F. D. Domingues, F. Ceschin, R. Machnicki, M. A. Zanata Alves, P. L. de Geus, and A. Grégio, "Antiviruses under the microscope: A hands-on perspective," *Computers And Security*, 2022.