ARMED: How automatic malware modifications can evade static detection?

Raphael Labaca Castro, Corinna Schmitt, and Gabi Dreo Rodosek

Research Institute CODE
Bundeswehr University Munich, Germany
{Raphael.Labaca, Corinna.Schmitt, Gabi.Dreo}@unibw.de

***Abstract*—Modifying existing malicious software until malware scanners misclassify it as *clean* is an attractive technique for cybercriminals. In particular, fully automatizing the process can bring adversaries to generate faster effective threats. Recent studies suggest that injecting successful malware modifications could lead to corrupt executable files despite of detection. Therefore, we propose *ARMED* – Automatic Random Malware Modifications to Evade Detection – to bypass classifiers by automatizing valid malware generation based on detected threats. The goal is to understand how successful automatic perturbations can be used to avoid detection. In order to reach this goal, we take portable executable malware and add a number of small random injections to evade detection without affecting the malware structure. Our experiments proved that only six perturbations are required to create new functional malware samples exhibiting exactly the same behavior yet with up to 80% less detections based on original malware that was previously detected. We show that within a few minutes an adversary could take a previously detected malware and convert it in a clean new mutation bypassing static malware scanners.**

***Index Terms*—ARMED framework, malware, byte-level perturbations, evasion**

I. Introduction

Malware creation or propagation belong to the routine of almost every attacker on the Internet and it has been studied for over 30 years since computer viruses have been formally introduced [1]. It is an important resource for cybercriminals, who pursue their own benefit through illegal activities (e.g., stealing sensitive information or encrypting it and holding it hostage for ransom). In parallel, researchers, malware analysts, and security companies spend most of the time researching and improving solutions in order to protect information against malicious software [2] [3]. Furthermore, users have also the obligation to protect themselves by using their systems in a safer manner and updating software frequently [4]. However, proactive defense is a challenge. Protecting users from unknown threats often leads to an arms race between malware authors and security measures. Although defense mechanisms have strongly improved in the last decades, attackers are still able to find new ways to compromise their victims without being detected either by finding novel approaches, using complex packers, exploiting vulnerabilities, or social engineering victims.

Despite of how malicious campaigns are targeting victims, it generally involves a piece of malware to break into the system to fulfill its goals such as exfiltrating data or attempting to render a system unusable [5]. However, detected malware is generally useless for new attacks and therefore automatizing malware generation by recycling detected threats is an interesting approach for criminals and an additional constraint for malware protections. As the number of malicious software increased in the last years to hundreds of thousands of new malware samples daily [6], security technologies can no longer purely rely on signatures anymore, instead they started using heuristics and machine learning techniques for over a decade [7]. The problem is that exclusively relying on machine learning can be accurate but it is hardly recommended given the large number of false positives (FP) that may arise [8]. That means, finding the right balance can be challenging when dealing with automatically generated malware mutations, which can either bypass static malware detection or generate a high-rate of FP to more sensitive malware scanners.

In order to understand how static malware detection can behave against automated generated malware mutations, we propose the following three research questions that inspired our solution ARMED (**A**utomatic **R**andom **M**alware Modifications to **E**vade **D**etection): i) How automatic random binary-level modifications in malware can affect static malware detection. ii) How complex and expensive is the generation of functional versus non-functional mutations. iii) Which influence does the number of perturbations injected have to functionality and evasion of new mutations.

With these three questions in mind ARMED was created including a three-step process to generate new malware mutations (modified versions of the original file) with high evasion rate compared to the original malware given as input and support the understanding of malware structure to improve defense strategies. First, the manipulation box modifies the original malware by inserting random binary-level perturbations. Second a sandbox (oracle) tests its functionality to make sure the new mutation is not corrupt and third malware scanners report detections for the new mutation. The detection rate of the latter is compared to that of the original input file and depending whether the results are successful (detection rate decreased) new mutations are stored in different databases (cf. Figure 1).

The rest of this paper is structured as follows: Section II

provides the related work and knowledge that inspired design decisions in Section III. In Section IV we introduce the ARMED framework and evaluate its results in Section V. Finally, we conclude our paper in Section VI.

II. Background

This section lays the groundwork for the design of ARMED in Section III by introducing the structure of input files used, malware detection strategies, and evasion techniques.

A. Portable executable files

Portable executable (PE) files were created to establish a way for the Microsoft Windows operating system to run applications and to store the important data needed during its execution [9]. The PE format includes executables extensions such as exe, dll, and object code. PEs are an extension of Common Object File Format (COFF) [10], which is a format for executables previously introduced on Unix systems. Editing PE files has been of interest for many years but the development of such tools are complex and expensive at the same time [11] [12]. In addition, many authors implement their own parser that is linked to a specific language. Thus, a recently approach has been introduced, dubbed LIEF [13], which is developed to provide a cross platform library that parses and modifies different executable formats including PE. Despite it can have limitations to parse some files where code needs to be patched after building the executable, it provides a useful interface to interact with PE software.

B. Malware detection

Malware detection can be implemented in several ways but most malware scanners use a combination of different technologies like *signature-based*, *anomaly-based* and *heuristic-based* techniques. *Signature-based* methods [14] rely on identifying files using cryptographic hash values of the binary or specific string inside the file and comparing it to known signatures in a database. This technique is very fast, though it prevents new variants of malware to be correctly identified. *Anomaly-based* methods [15] focus on identifying what a normal process looks like and attempts to classify malware based on its activity rather than patterns. That is particularly useful when malware started to be generated much faster than analysts were able to deliver signatures. *Heuristic-based* approaches [16] attempts to match behavior of a malware sample with previously defined actions. This technique is often combined with others as supplementary detection. In addition, artificial intelligence and machine learning [17] became an important approach of detection methods and have been widely used to classify malware [18] with more recent approaches including neural networks [19] [20] and deep convolutional networks [21].

Another classification of detection strategies can be done by grouping approaches into statically and dynamically. Static approaches focus on classifying samples by analyzing it without executing the file and, for example, extracting information about strings while dynamic approaches attempt to analyze its behavior in runtime in a controlled environment, which is useful to understand, among others, API calls. Both approaches have its limitations [22] and good analysis will probably leverage both techniques [23]. However, given the number of malware samples produced daily and the ability to detect them prior to execution, static malware detection still plays a key role in computer security and therefore generating mutations that can no longer be detected based on previously known malware remains an important issue if automatization of the process is successfully achieved.

C. Malware evasion

Obfuscation techniques [24] attempt to hide software's functionality and it can be a strong tool for malware evasion. $Packers$ are applications that modify the execution form of executable files while keeping their original functionality. These are specially designed to obfuscate software in order to hide its behavior from third-parties like Themida or Armadillo [25]. Currently, they are also widely used to make more complex the analysis and reverse engineering of malware but were initially developed [26] to improve efficiency of memory and bandwidth used when storing and transferring files. After these resources became more accessible, commercial vendors still continued to use $packers$ to deliver their products in order to protect their software. However, criminals have also started to use it to bypass detection. In fact, $packers$ are developed by both corporate vendors and malicious organizations. Further research developed online services like PolyPack [27] that offers an array of $packers$ and malware scanners to test against each other in order to find which $packers$ provided the best evasion rate. Results showed that their approach outperforms the best packer, Themida, by 40%. Another technique is $malware\ diversification$, which provides a way of generating an almost infinite number of binaries with the same functionality but very low similarity [28]. Further approaches extended the Low Level Virtual Machine (LLVM) implementation by replacing instructions and types of control-flow randomization [29]. $Polymorphism$ [30] is another mechanism used to attempt evasion. The malware code implements a mutation method that modifies its content using an encryption key. Each time the malware runs, a mutation happens using a different key, which provides a new copy of the code that forces a new signature by scanners. However, the polymorphic engine deciphers a similar payload code and load it into memory each time and that makes a post-execution detection using signatures more feasible. Consequently, there is a similar technique called $metamorphism$ [31]. In this case when the malware runs, the code that is loaded into memory is the one that changes and it writes back to the system a new version of the malware. That technique helps bypass detection – mostly signature-based – at some extent but writing this kind of malware could be complex and expensive [32].

III. Design Decisions

With research questions and related work in place scenario, requirements, and assumptions can be specified leading to the

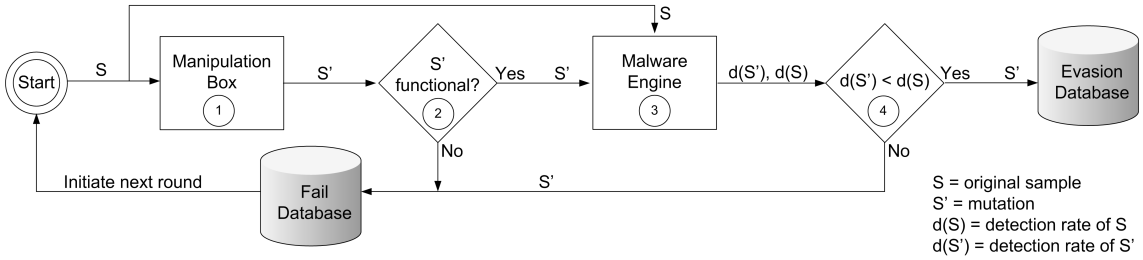

Fig. 1. ARMED framework

final architecture of ARMED and the implemented workflow.

A. Scenario

Generally, detection engines are sensitive and require permanent learning to detect new malicious files successfully. From an attacker's perspective the detection rate of such engines should be low and, thus, modifications are required to render previously-known malware undetected again. In order to reach this goal different ways of bypassing detection exist as described in Section II. The strategy of injecting perturbations to modify the PE file's structure is applied to compromise static analysis detection approaches and lead to misclassifying new malware mutations.

The nature of malware creation is complex and there is a plethora of malware samples being created daily [6]. We focus on using as input those samples, which have been previously known and reported to all engines listed in VirusTotal. The current detection rate of the original file is used as benchmark to measure against new mutations' detection rates and compare their performance.

B. Requirements

We introduce four requirements that we consider essential for ARMED as they will make sure the framework generates functional mutations:

- **Automatization**: Malware attackers have different tools to create their new samples and campaigns. However, a part of the job still needs to be adjusted manually. We are aiming to understand how automatic solutions could be created to bypass malware engine detections based on previously known malware.
- **Platform**: Although our framework allows to inject perturbations on different types of executable files, we will be focusing on Windows PE malware as this constitutes, in our opinion, a large portion of currently propagated malware given that Microsoft's operating systems are predominantly used.
- **Size**: We used malware samples of at least 100 kilobytes and maximum 1 megabyte to keep perturbations feasible and also data transfer efficient between our platform and the services used. However, the framework supports larger files, too.
- **Functionality**: Given that perturbations could affect the basic structure of a portable executable file, we need to ensure that any mutation will be functional before testing its detection rates.

C. Assumptions

We assume that currently most of malware campaigns focus on 32/64 bits Windows platforms. Thus, we use Win32/64 PE files as input for ARMED.

Detection rates come from VirusTotal as it is a commonly used and free available tool for detection of malicious files. We are aware that it is a platform grouping most of anti-malware solutions and should not be used to compare results among comercial solutions [33]. In fact, benchmarking malware scanners has been long discussed about how to implement a neutral test environment that accurately reflects detection rates [34] [35]. That is given the fact that many security solutions have additional features and layers that are not being implemented by online platforms. In order to have more accurate results for any individual malware scanner it would be recommended to install and configure it separately and assess the results for that engine. That approach could be interesting for targeted attacks where the goal is to bypass specific software.

D. Notations

An unmodified sample is used as input for ARMED and is called $original\ sample\ S$. A $perturbation\ p$ is defined as a specific modification injected into any given sample resulting in a modified sample S' called $mutation$. Perturbations are byte-level modifications that include changes from appending random characters to renaming or adding new sections to the malware structure. A set of perturbations is called $sequence$ and two or more sequences can have the same length but different order of perturbations performed. That means, even by keeping the same input sample S the resulting modified sample S' can differ all the time depending on the applied sequence. An $oracle$ is defined as an external service ensuring the file (S' or mutation) is functional (valid) and not corrupt (invalid).

The $detection\ rate\ d$ is defined as the rate of engines detecting the sample (either S or S') as malicious and is stored in a database including information of the applied sequence and a link to detection webpage with engine results. Each time ARMED initiates with a sample S a new round r starts.

Combining the above introduced terms means, that when ARMED is performed with r=3, the same initial sample S

Algorithm 1 The automatic generation of mutations

1: *M: set of malware files*
2: *M*: set of evasive samples*
3: *p: number of perturbations to inject*
4: **for each** $r \in rounds$: **do**
5: *sample a batch of* M
6: $S \leftarrow random(M)$
7: $S' \leftarrow ManipulationBox(S, p)$
8: $a(S') \leftarrow analysis(S')$
9: **if** $a(S') \in$ *Functional:* **then**
10: $d(S), d(S') \leftarrow detection(S, S')$
11: **if** *d(S') < d(S):* **then**
12: **update** *M* with S'*
13: **end if**
14: **end if**
15: **end for**

can be used and will generate three different modified samples S'. The resulting mutations will likely have different detection rates d. For each round r a report is stored in the database.

IV. ARMED Framework

Based on defined requirements and assumptions the ARMED framework can be specified. We implemented our solution building on the work from Anderson et al. [40], which is an extension from the OpenAI gym environment [41].

The following three components – manipulation box, sandbox (oracle), and malware scanner (cf. Figure 1) – are briefly described in ARMED's architecture in Section IV-A. Additionally, a database is required to store samples, reports, and detection rates for further analysis. The resulting workflow using this architecture is presented in Section IV-B following Algorithm 1.

A. Architecture

Figure 1 illustrates the handling of an original sample S triggering ARMED to develop a mutation S'. During this modification process the sample passes through the following components in a row:

The **manipulation box** is in charge of injecting random perturbations generating a mutation S'. As defined in Section III a finite number of perturbations can be performed building a sequence that can differ on each round.

The **sandbox** is a virtual machine service [42] used to statically and dynamically analyze the mutation S'. In this case, it proves functionality in order to make sure that integrity is kept. The sandbox provide also malicious indicators including API calls and behavior of the file, file dropped, screenshots and traffic dumps. These return a maliciousness indicator that is also taken into account to determine how successful the new mutation is. We assume that malicious actors would have enough time to run multiple attempts until finding executable samples S', hence functionality comes at the expense of time.

The **malware scanner** is used to determine detection rates for S and S'. We decided to use an online service for testing common used detection engines in parallel. An attacker could use the same strategy and, thus, it is recommended. However, we are aware of the limitations [33] of command-line versions and the possibility that some engines do not behave exactly the same on desktop versions or the fact that parametrization can affect aggressiveness of detection.

B. Workflow

ARMED's workflow follows Algorithm 1 that can be broken down into four steps, as observed in Fig 2, including the previously mentioned three components of the architecture.

Step 1 is initiated as soon as a sample S (a PE file) is handed over to the manipulation box. From 11 defined [40] perturbations implemented via LIEF[1] we decided to use nine, namely: $overlay\_append$, $section\_rename$, $section\_add$, $section\_append$, $remove\_signature$, $remove\_debug$, $break\_optional\_header$, upx_pack, and upx_unpack. The following ($imports\_append$ and $create\_entry$) are not used, as experiments showed that they may fail to patch the sample properly and most likely create corrupt files given entry point errors. The perturbations are randomly chosen and injected at binary-level (line 7) generating the mutation S'.

In order to generate valid samples every new S' needs to be tested against an oracle to make sure the newly created samples are still valid. This test builds **Step 2** of ARMED and is caused by line 8. In case S' passes the test successfully, it can be guaranteed that the perturbations did not break the structure of PE files. In this step ARMED is flexible in which tool is used for the functionality test. The only requirement is to have an interface or API available, either remotely or locally, to where each request can be sent. After submitting the mutation to the sandbox it will be executed and analyzed returning all relevant information in a JSON report including: persistence, fingerprint, stealthiness, and overall malicious indicators as well as networking information (e.g., DNS requests, domains/servers contacted, and potential URLs in memory). Those fields are then parsed and used to determine whether a sample was able to be executed. In case the functional test fails in step 2 the information is stored in a so-called "Fail Database" and another ARMED round initiates keeping S and applying another sequence in step 1.

Assuming step 2 was passed successfully malware engines are contacted in **Step 3** to determine detection rates (line 10). Here S and S' are sent for detection. Therefore, an API was implemented allowing to submit samples and retrieve detection reports ($d(S)$ and $d(S')$). For comparing the detection rates received it is important to note hat the number of maximum engines may vary from one detection to another given that each analysis is independent and sometimes not all engines are available, hence they are normalized in the final result.

Finally, in **Step 4** the detection rates received are compared (line 11) to match the overall goal: $d(S') < d(S)$, which means

[1]The goal of the library LIEF is to provide a cross-platform bridge to manipulate format internals of a PE file.

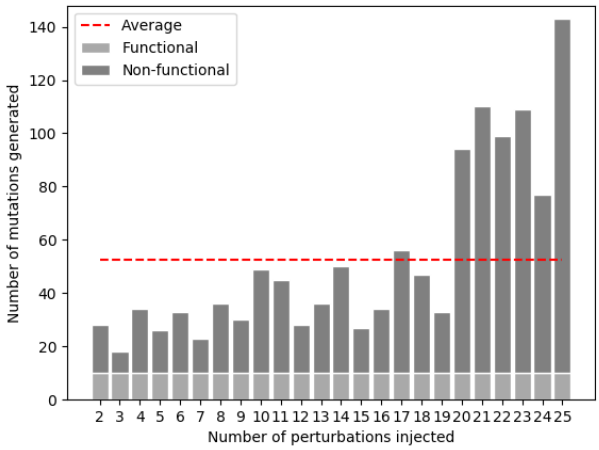

Fig. 2. Sample S_a: Functional vs. non-functional mutations (n=1265)

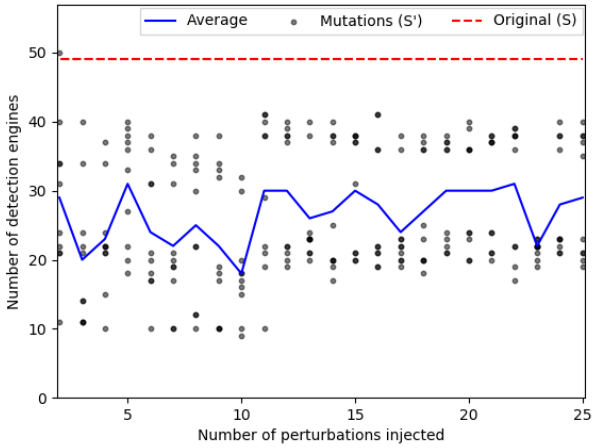

Fig. 3. Sample S_a: Detection of 240 mutations based on perturbations injected

the manipulation box performed successful, producing a valid PE file S' that has a lower detection rate as the initial sample S. Thus, S, S', along with a report including perturbation sequences, $d(S)$, $d(S')$, and the URLs corresponding to the detection result overview are stored in an "Evasion Database". If $d(S'))$ >= $d(S)$ the entire process failed in step 4, the information is stored in "Fail Database" and a new round r starts with step 1 using same initial sample S until a successful S' is found.

V. Evaluation

The ARMED framework is evaluated concerning functionality, detection quality, sequence structure, and processing time. Windows PE files, classified as malware, are used as input. Several rounds of ARMED are performed injecting different sequences of perturbations to generate a mutation sample causing a lower detection rate than the initial sample. Finally, limitations of ARMED can be identified leading to optimizations in the future.

A. Functionality

In order to evaluate the functionality of ARMED a set of Windows PE malware samples were tested generating hundreds of valid mutations. Randomly selected original samples were used to generate thousands of new malware mutations and analyze the results as follows. It could be observed that some of the PE samples tested returned error when parsed by LIEF because the library does not use any disassembler causing a corruption during the import of LEA instructions[2] that are implemented on the import address table. In order to overcome this problem, an initial pre-filtering is needed to isolate functional samples that are correctly parsed. Not all mutations generated are valid since perturbations can make the file corrupt. Our experiments showed that we usually need to create at least the double of mutations than the number of valid files we intend to reach.

As Figure 2 illustrates, every column shows the number of mutations needed to be generated in order to achieve 10 valid mutations, meaning they are passing the functionality test successfully. For instance, for all number of perturbations injected we needed to generate at least 20 mutations for every case in order to reach 10 valid, except for p = 3 where 18 mutations were enough to generate 10 functional ones. On the other hand, for mutations with 20 perturbations (p = 20) we needed to generate 94 files, more than nine times the number of mutations expected in order to achieve the functional mutations desired. And from 20 and onwards, we can observe that the more perturbations injected the frequent mutations are built corrupt and, thus, it takes much longer to achieve valid malware mutations. The last case analyzed, p = 25, generated 133 corrupt mutations until 10 functional were finally found. The average is at 52.7 as it is marked by the red line in Figure 2. This behavior repeated itself with small variations across different samples as we can observe in Figure 4. In this case, we used another original sample S to generate mutations. Again, for each case more than double of files needed to be generated to find 10 valid mutations.

Those that received five perturbations only needed to be generated 15 times, whereas for p = 18 108 mutations were required. The average in this case was at 55.2. That is an interesting fact to keep in mind when using implementations without oracles to confirm whether new mutations are valid files. Thus, functionality testing or using an oracle to determine whether a mutation is executable is an expensive but very necessary approach to make sure adversarial examples are valid. Otherwise, the system will risk to generate evasive non-viable mutations, which fail to fulfil its purpose.

B. Detection

In order to test the detection criteria for ARMED the same original samples S_a and S_b were used, which were detected by

[2]Load Effective Address (LEA) instructions are arithmetic operations designed to map high level memory references.

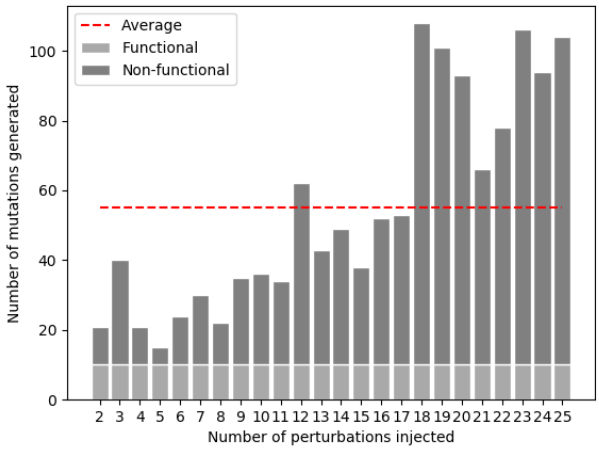

Fig. 4. Sample S_b: Functional vs. non-functional mutations (n=1325)

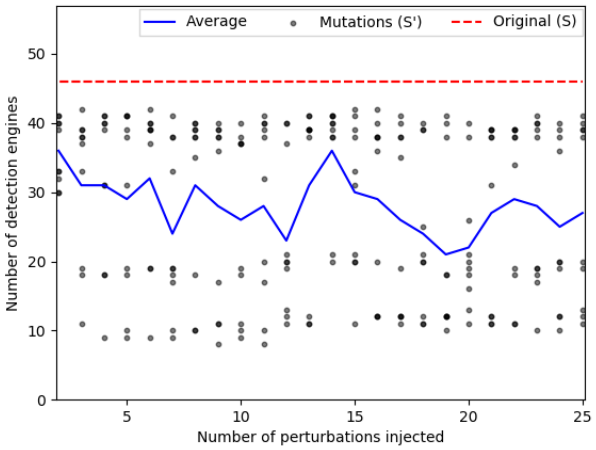

Fig. 5. Sample S_b: Detection of 240 mutations based on perturbations injected

49 and 46 engines respectively. We generated for each original sample S 240 successful mutations S' divided across 24 groups. Each group accounts for a sequence of perturbations of length starting with two until 25. We took 24 cases to compare against each other and see how different mutations perform versus engine detections. The reason we stopped collecting at that number is basically because higher perturbation rates do not seem to improve detection results.

Moreover, in our experiments we tested injecting successfully until 50 perturbations without affecting the new mutation's functionality, nonetheless for that number of perturbations the number of mutations that need to be generated in order to achieve successful mutations increased strongly rendering the whole process much slower. Even after 20 perturbations the mutations do not seem to be more successful in minimizing detection, which makes larger sequence vectors not necessarily more efficient. We also experimented with high number of perturbations – between 50 and 500 – and although the samples were not functional anymore, it proved us that mutations can be more or less successful despite the number of perturbations inserted (cf. Table I). In fact, none of the tests reached complete evasion and the lowest detection rates were around four to five engines for corrupt files with 50 and 500 injections respectively while valid mutations reported five to six detections with only five perturbations inserted.

Therefore, our results proved that a small number of perturbations can have a significant impact on new mutations that would exploit most of engines' static detections' sensitivity.

Furthermore, with the sample observed in Figure 3 we tested sending half of the mutations to VirusTotal [33] and waiting a few weeks to continue the test with the rest. As we can observe, good evasion results (around 10 engines or less) were no longer achieved after the pause. That implies that after a few days that the new mutations have been uploaded, most of the engines start to detecting them as we can be seen with all mutations between p = 11 and p = 25. On the other hand, for another original sample S, we generated all mutations within a couple days and analyzed the results as showed in Figure 5. In this case, unlike the previous sample, detection rate around 10 was observed throughout all cases. Moreover, interesting evasion results were observed for mutations with p = 9 and p = 11 where only 8 out of 68 engines positively detected the malware. That leads us to believe that more successful propagation campaigns should most likely refrain from uploading malware and instead test detection locally targeting specific engines.

TABLE I
COMPARING HIGH-EVASION MUTATIONS FROM SAME S (44/67 DETECTIONS)

Mutation	Perturbations	Detection Rate	Valid
S'_1	5	5/65	Yes
S'_2	5	6/66	Yes
S'_3	6	6/68	Yes
S'_4	50	4/68	No
S'_5	500	5/66	No

C. Sequence Structure

As defined in Section III a sequence has a length and a specific order of perturbations. In order to analyze the impact of the sequence structure, the detection results from the experiments with different sequences were plotted in Figures 3 and 5. The y-axis gives the number of engines detecting S' as malicious and the x-axis determines the sequence length (= number of perturbations). As a benchmark the detection rate of the original sample S is included as red dashed line. Dots indicate the respective detection rate for each S'. The blue line indicates the average of the detections per sequence. Figure 3 shows that for different mutations the average rate of detections usually oscillates between 20 and 30 engines, whereas in Figure 5 some of the mutations are detected in the

range of 30 and 40 engines. This shows that some original samples S generate more heterogenous variants of S'.

A specific number of perturbations produced better results than others (e.g., p = 10 in Figure 3 and p = 9 in Figure 5). Looking to the overall detection rates of S' in Figure 5 it can be stated that with almost all number of perturbations a small number of detections, ergo high number of evasion, can be achieved indicating that the perturbation order in the sequence may be more relevant than the sequence length itself. For small number of perturbations, it can also be observed that some sequences might result in higher detection rates that the benchmark (e.g., p = 2 in Figure 3). However, these outliers seem to be connected to the fact that after a while more engines detect the original S and the less S' is changed (smaller p values) the more similar to the original sample they are and malware scanners adjust their detections better thus returning higher detection rates for very similar mutations.

D. Processing Time

The entire ARMED workflow, triggered by an initial sample S and creating S' where $d(S') < d(S)$, takes currently around 5 minutes in average. Performing step 1 takes less than a second and step 2 between 30 and 45 seconds. The most demanding part of the process is step 3, as third-party services are involved and performance is highly influenced by availability of the systems and probably the amount of people using the service at the same time. The processing time for step 3 can be drastically reduced if there is a surrogate classifier, which can be used to probe S' and then transfer knowledge to real-world classifiers. In our experiments we also evaluated MetaDefender [43], a service similar to VirusTotal that claims to be much faster but problems with license agreement and frequent FPs in results refrained us to continue to pursue that path. Moreover, we implemented three scanners that are commercially available to optimize step 3 and we were able to reduce processing time from 5 minutes to 20 seconds in average. That would make the whole process successful in around one minute, yet we kept reports from the online service as more than 60 engines are compared instead of three.

E. Limitations

Based on the evaluation results presented throughout Sections V-A to V-D limitations were identified concerning collisions and detections. Our experiments showed that when the sequence injected is of length one, or some cases two, many mutations end up being duplicates of the original sample S. That means, the cryptographic hash calculated for both files are equal. Therefore, we skipped p = 1 and started carefully considering mutations with sequences of length two to 25, which gives us 24 different cases to analyze how mutations perform against malware classifiers.

Security layers can help preventing the infection even though S' will be less detected. For instance, if S' is generated from a malware dropper or downloader, the download of the malware or the payload dropped can be detected by firewall configurations or the new sample might be already known for the malware engine. Therefore, ARMED can be implemented across different malware samples (e.g., droppers, downloaders and payload) and combined with additional layers of evasion such as $packers$ in order to increase the evasion rate.

VI. Conclusions & future work

In this paper, we introduced the ARMED framework to inject random perturbations on Windows PE malware files in order to create mutations S' with optimal evasion among common malware engines. ARMED will be useful for researchers and analysts to understand how attackers could automatically manipulate malware in order to bypass static detection classifiers. Evaluations showed that random binary-level modifications, resulting in S', can decrease malware engines' detection rate up to 80% comparing to the detections of the original sample S. Generating functional mutations depends largely on the specific order of perturbations randomly injected. In average around 18% of mutations generated in our experiment are functional and not corrupted. Increasing functionality rate remains an open issue to be addressed. High evasion can be achieved with both, small and large sequences. Nevertheless, it was acknowledged that if the sequence length is beyond 20, it is challenging to achieve functionality, which means a higher number of S' must be generated to achieve a minimum number of functional mutations at the end.

As future work it is considered to improve the understanding of the perturbations' order using machine learning approaches. Given that currently the sequence is randomly chosen, we intend to further develop techniques to define successful injections and predict optimal evasion mutations depending on characteristics of the input sample S.

Acknowledgment

The authors would like to thank VirusTotal for their collaboration as well as the Chair for Communication Systems and Network Security and the research institute CODE for their comments and improvements.

References

[1] F. Cohen, "Computer Viruses," Ph.D. dissertation, Department of Electrical Engineering, University of Southern California, Los Angeles, CA, USA, January 1986.

[2] F. Cohen, "Models of practical defenses against computer viruses," in *Computers & Security*. Elsevier, 1989, vol. 8, no. 2, pp. 149 –160.

[3] J. O. Kephart, G. B. Sorkin, W. C. Arnold, D. M. Chess, G. J. Tesauro, S. R. White, and T. J. Watson, "Biologically Inspired Defenses Against Computer Viruses," in *14th International Joint Conference on Artificial Intelligence*, ser. IJCAI. New York, NY, USA: ACM, August 1995, pp. 985–996.

[4] National Research Council and Committee, *Computers at Risk: Safe Computing in the Information Age*. Washington, DC, USA: National Academies Press, March 1991, vol. 2.

[5] R. Herold and M. K. Rogers, *Encyclopedia of Information Assurance*. Boca Raton, FL, USA: Auerbach Publications imprint of CRC Press, 2010, vol. 4.

[6] Symantec Cooperation, "2018 Internet Security Threat Report, Volume 23," https://www.symantec.com/security-center/threat-report, March 2018, last access Aug. 31, 2018.

[7] WeLiveSecurity, "Is machine learning cybersecurity's silver bullet?" https://www.welivesecurity.com/wp-content/uploads/2017/08/NextGen_ML.pdf, 2017, last access Sep. 13, 2018.

[8] D. Gavrilut, R. Benchea, and C. Vatamanu, "Optimized Zero False Positives Perceptron Training for Malware Detection," in *14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, ser. SYNASC. New York, NY, USA: IEEE, September 2012, pp. 247–253.

[9] M. Pietrek, "Peering Inside the PE: A Tour of the Win32 Portable Executable File Format," https://msdn.microsoft.com/en-us/library/ms809762.aspx, March 1994, last access Aug. 10, 2018.

[10] M. Pietrek, "PE Format," https://docs.microsoft.com/en-us/windows/desktop/Debug/pe-format, 2018, last access Aug. 28, 2018.

[11] J. R. Larus and E. Schnarr, "EEL: Machine-independent Executable Editing," in *SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI, vol. 30, no. 6. New York, NY, USA: ACM, June 1995, pp. 291–300.

[12] A. Srivastava and D. W. Wall, "A Practical System for Intermodule Code Optimization at Link-Time," in *Journal of Programming Languages*, vol. 1, no. 1. Noida, India: STM Publication, December 1992, pp. 1–18.

[13] R. Thomas, "LIEF - Library to Instrument Executable Formats," https://github.com/lief-project/LIEF, 2017, last access Aug. 22, 2018.

[14] I. A. Saeed, A. Selamat, and A. M. Abuagoub, "A Survey on Malware and Malware Detection Systems," in *International Journal of Computer Applications*. New York, NY, USA: Foundation of Computer Science, April 2013, vol. 67, no. 16, pp. 25–31.

[15] H. S. Teng, K. Chen, and S. C. Lu, "Adaptive Real-time Anomaly Detection Using Inductively Generated Sequential Patterns," in *Computer Society Symposium on Research in Security and Privacy*. New York, NY, USA: IEEE, May 1990, pp. 278–284.

[16] Z. Bazrafshan, H. Hashemi, S. M. H. Fard, and A. Hamzeh, "A Survey on Heuristic Malware Detection Techniques," in *5th Conference on Information and Knowledge Technology*, ser. IKT. New York, NY, USA: IEEE, May 2013, pp. 113–120.

[17] I. Firdausi, A. Erwin, A. S. Nugroho *et al.*, "Analysis of Machine Learning Techniques Used in Behavior-based Malware Detection," in *2nd International Conference on Advances in Computing, Control and Telecommunication Technologies*, ser. ACT. New York, NY, USA: IEEE, December 2010, pp. 201–203.

[18] M. G. Schultz, E. Eskin, F. Zadok, and S. J. Stolfo, "Data Mining Methods for Detection of New Malicious Executables," in *Symposium on Security and Privacy*, ser. S&P. New York, NY, USA: IEEE, May 2001, pp. 38–49.

[19] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas, "Malware detection by eating a whole exe," in *arXiv preprint arXiv:1710.09435*, 2017.

[20] E. Raff, J. Sylvester, and C. Nicholas, "Learning the PE Header, Malware Detection with Minimal Domain Knowledge," in *10th Workshop on Artificial Intelligence and Security*. New York, NY, USA: ACM, October/November 2017, pp. 121–132.

[21] M. Krčál, O. Švec, M. Bálek, and O. Jašek, "Deep Convolutional Malware Classifiers Can Learn from Raw Executables and Labels Only," in *6th International Conference on Learning Representations*, ser. ICLR, Vancouver, BC, Canada, April/May 2018, pp. 1–4.

[22] A. Moser, C. Kruegel, and E. Kirda, "Limits of Static Analysis for Malware Detection," in *23rd Annual Computer Security Applications Conference*, ser. ACSAC. New York, NY, USA: IEEE, December 2007, pp. 421–430.

[23] A. Damodaran, F. Di Troia, C. A. Visaggio, T. H. Austin, and M. Stamp, "A Comparison of Static, Dynamic, and Hybrid Analysis for Malware Detection," in *Journal of Computer Virology and Hacking Techniques*. Heidelberg, Germany: Springer, February 2017, vol. 13, no. 1, pp. 1–12.

[24] I. You and K. Yim, "Malware obfuscation techniques: A brief survey," in *Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on*. IEEE, 2010, pp. 297–300.

[25] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "Rambo: run-time packer analysis with multiple branch observation," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. San Sebastian, Spain: Springer, 2016, pp. 186–206.

[26] P. O'Kane, S. Sezer, and K. McLaughlin, "Obfuscation: The hidden malware," in *IEEE Security & Privacy*. IEEE, 2011, vol. 9, no. 5, pp. 41–47.

[27] J. Oberheide, M. Bailey, and F. Jahanian, "PolyPack: An Automated Online Packing Service for Optimal Antivirus Evasion," in *3rd USENIX Conference on Offensive Technologies*, ser. WOOT. San Diego, CA, USA: USENIX Association, August 2009, pp. 1–6.

[28] M. Payer, "Embracing the New Threat: Towards Automatically Self-diversifying Malware," in *The Symposium on Security for Asia Network*, Singapore, Singapore, April 2014, pp. 1–5.

[29] M. Payer, S. Crane, P. Larsen, S. Brunthaler, R. Wartell, and M. Franz, "Similarity-based matching meets malware diversity," in *Preprint arXiv:1409.7760*. arXiv, 2014.

[30] W. Wong and M. Stamp, "Hunting for Metamorphic Engines," in *Journal in Computer Virology*. Heidelberg, Germany: Springer, December 2006, vol. 2, no. 3, pp. 211–229.

[31] Q. Zhang and D. S. Reeves, "Metaaware: Identifying metamorphic malware," in *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*. IEEE, 2007, pp. 411–420.

[32] E. Konstantinou and S. Wolthusen, "Metamorphic virus: Analysis and detection," in *Royal Holloway University of London*, 2008, vol. 15, p. 15.

[33] VirusTotal, "VirusTotal Frequently," https://www.virustotal.com/, 2018, last access Oct. 10, 2018.

[34] S. Gordon and R. Ford, "Real world anti-virus product reviews and evaluations-the current state of affairs," in *Proceedings of the 1996 National Information Systems Security Conference*, 1996.

[35] M. Christodorescu and S. Jha, "Testing malware detectors," in *ACM SIGSOFT Software Engineering Notes*. ACM, 2004, vol. 29, no. 4, pp. 34–44.

[36] M. Matsumoto and T. Nishimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 8, no. 1, pp. 3–30, 1998.

[37] T. E. Hull and A. R. Dobell, "Random number generators," *SIAM review*, vol. 4, no. 3, pp. 230–254, 1962.

[38] M. Matsumoto and Y. Kurita, "Twisted gfsr generators," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 2, no. 3, pp. 179–194, 1992.

[39] P. L'Ecuyer and R. Simard, "Testu01: Ac library for empirical testing of random number generators," *ACM Transactions on Mathematical Software (TOMS)*, vol. 33, no. 4, p. 22, 2007.

[40] H. S. Anderson, A. Kharkar, B. Filar, D. Evans, and P. Roth, "Learning to Evade Static PE Machine Learning Malware Models via Reinforcement Learning," in *Computing Research Repository*. Ithaca, NY, USA: Cornell University, January 2018, pp. 1–9.

[41] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[42] Cuckoo, "Cuckoo sandbox," https://cuckoosandbox.org, 2018, last access Nov. 11, 2018.

[43] MetaDefender, "Metadefender: A More Private Alternative to VirusTotal," https://www.opswat.com/blog/metadefender-more-private-alternative-virustotal, 2018, last access Sept. 21, 2018.