Tutorial: An Overview of Malware Detection and Evasion Techniques

Fabrizio Biondi[1(✉)], Thomas Given-Wilson[2], Axel Legay[2], Cassius Puodzius[2], and Jean Quilbeuf[2]

[1] CentraleSupélec/IRISA, Rennes, France
fabrizio.biondi@inria.fr
[2] Inria, Rocquencourt, France
{thomas.given-wilson,axel.legay, cassius.puodzius,jean.auilbeuf}@inria.fr

Abstract. This tutorial presents and motivates various malware detection tools and illustrates their usage on a clear example. We demonstrate how *statically-extracted syntactic signatures* can be used for quickly detecting simple variants of malware. Since such signatures can easily be obfuscated, we also present *dynamically-extracted behavioral signatures* which are obtained by running the malware in an isolated environment known as a *sandbox*. However, some malware can use sandbox detection to detect that they run in such an environment and so avoid exhibiting their malicious behavior. To counteract sandbox detection, we present *concolic execution* that can explore several paths of a binary. We conclude by showing how *opaque predicates* and *JIT* can be used to hinder concolic execution.

1 Introduction

Context. Malicious software known as *malware* is a growing threat to the security of systems and users. The volume of malware is dramatically increasing every year, with the 2018 Cisco report estimating a 12 times increase in malware volume from 2015 to 2017 [5]. For this reason, effective and automated malware detection is an important requirement to guarantee system safety and user protection.

Malware signatures. Signature-based malware detection refers to the use of distinctive information known as *signatures* to detect malware. An extraction procedure is performed on an unclassified binary file to extract its signature, this signature is then compared against similar signatures of malware to determine whether the unclassified binary's signature indicates malicious behavior. The simplest kind of signatures are *syntactic signatures* [28,30] that detect malware based on syntactic properties of the malware binaries (like their length, entropy, number of sections, or presence of certain strings). Alternatively, *behavioral signatures* [28,30] can be based on behavioral properties of malware (like their interaction with the system and its network communications).

T. Margaria and B. Steffen (Eds.): ISoLA 2018, LNCS 11244, pp. 565–586, 2018.
https://doi.org/10.1007/978-3-030-03418-4_34

Static analysis. Syntactic signatures can be easily extracted from binaries using *static analysis* [28], i.e. analyzing the binary without having to execute it e.g. by disassembling the binary or by scanning it for specific strings. Importantly, malware detection based on statically-extracted syntactic properties is in practice the only technique fast enough to be used for on-access malware detection, which is why antivirus software are typically based on this type of analysis. However, obfuscation techniques [16] exist that modify the binary code to change its syntactic properties and make it harder to analyze by static analysis while keeping the same behavior. We present static malware detection based on syntactic signatures in Sect. 2.

Dynamic analysis. Due to the weaknesses of static signatures, behavioral signatures are used to counter obfuscation techniques that change the malware's syntactic properties but not its behavior. A common technique to analyze a binary's behavior is *dynamic analysis* [28] consisting of executing the malware and observing its effects on the system. To avoid infecting the analyst's system and to prevent the malware from spreading, the malware is commonly executed in a *sandbox*, i.e. a protected and isolated environment that has been instrumented to be easy to analyze and restore after infection. However, malware can implement sandbox detection techniques to determine whether they are being executed in a sandbox, in which case the malware avoids exhibiting its malicious behavior and often delete itself. We present dynamic malware detection based on behavioral signatures in Sect. 3.

Concolic analysis. The main limitation of dynamic analysis is that it extracts and analyzes only one of the possible execution paths of the analyzed binary, e.g. the one that avoids exhibiting malicious behavior. To address this limitation, *concolic analysis* (a portmanteau of CONCrete symbOLIC) [9,25] has been developed to extract a binary file's behavior while covering as many of the binary's possible execution paths as possible. Concolic analysis maintains a symbolic representation of the constraints found during its analysis, and relies on an external SMT solver to simplify such constraints and determine whether the possible paths can actually be executed or correspond to dead code. However, malicious techniques can be used to highly complicate the conditional constraints of the code, exponentially increasing the size of the symbolic representation and hindering concolic analysis. We present concolic malware detection based on behavioral signatures in Sect. 4.

For the sake of clarity and safety we will not work on a real malware. We provide in Fig. 1 a very simple C program that prints "I am evil!!!" to standard output, and we will treat this as malicious behavior. This simplification allows us to showcase various detection and obfuscation techniques in the rest of the paper. Unless otherwise stated, all examples are compiled using gcc with default settings, on an AMD64 GNU/Linux machine.

```c
#include<stdio.h>

int malicious_behaviour(){
  printf("I␣am␣evil!!!\n");
}

int main(int argc, char **argv){
  malicious_behaviour();
}
```

Fig. 1. The running example of fake malware written in C that will be used throughout the paper.

2 Static Analysis: Syntactic Pattern Matching

Syntactic signatures are used to classify binaries by looking at particular patterns in their code. Due to the simplicity of syntactic pattern matching, these techniques tend to be very fast in practice. In this section, we present the principles of syntactic pattern matching, then illustrate the approach with three different tools. We then explain how binaries can be obfuscated against such detection techniques and show a very simple case of obfuscation for our running example.

2.1 Principle

Signatures are defined by attributes and properties which describe some object of analysis. In the context of binary analysis, syntactic signatures refer to sequences of bytes that describe proprieties such as file checksum [1], type, API calls [11], etc.

For instance, once it is verified that a given binary follows the format of a Portable Executable (PE) file [21], other properties such as imported and exported functions, base addresses of the section headers, debug information, presence & features of standard binary sections, and physical & virtual addresses can be easily extracted due to the way the PE header format is defined.

This information can provide a rich understanding of how the binary is expected to run on the system as well as contextual information, such as the date on which the binary was (supposedly) compiled. However, as easily as this information can be extracted, it is also easy to modify or corrupt this information in order to mislead analysis [18].

To hinder syntactical analysis, an adversary can employ obfuscation techniques to conceal the syntactical properties of the original malware sample. Such techniques comprise simple ones such as *packing*, a technique to to compress the executable code as plain data and uncompress it only at runtime, or more advanced ones such as polymorphism and virtualization [26].

Despite their limitations [22], static syntactic signatures are largely employed in malware analysis. For example, ClamAV [6] allows the usage of syntactic signature in the YARA [23] format for protection against malicious files, and VirusTotal [32] provides an interface that takes YARA signatures to lookup matching files throughout its whole database.

Some of the biggest advantages of static signatures are the fact that they are very lightweight, cheap, and capable to capture architecture-dependent binary file attributes like binary section names and sizes. However, it is non-trivial to create and maintain syntactic signatures that: are specific enough to match only to the intended malware family; but not so specific that minor variants of the malware are not detected.

We now present some of the most popular tools for syntactic pattern matching including their signature formats, and discuss in more details their limitations.

2.2 Example Tool: PEiD

PEiD[1] is a tool for the detection of PE malware, packers, and compilers. Despite being already discontinued, PEiD is still largely used and sometimes updated by the users community.

PEiD defines an underlying grammar that allows the creation of new matching rules. This way, the inclusion of new rules to address a new malware, packer, or compiler does not depend on updating the tool and permits researchers to conveniently create and share rules.

As a first example, rules for .NET objects are displayed below.

```
[.NET DLL -> Microsoft]
signature = 00 00 00 00 00 00 00 00 5F 43 6F 72 44 6C 6C
4D 61 69 6E 00 6D 73 63 6F 72 65 65 2E 64 6C 6C 00 00 ??
00 00 FF 25
ep_only = false

[.NET executable -> Microsoft]
signature = 00 00 00 00 00 00 00 00 5F 43 6F 72 45 78 65
4D 61 69 6E 00 6D 73 63 6F 72 65 65 2E 64 6C 6C 00 00 00
00 00 FF 25
ep_only = false

[.NET executable]
signature = FF 25 00 20 40 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00
ep_only = true
```

Each rule starts with a string identifier between square brackets, which is displayed to the user whenever the rule is matched. The `signature` line contains the signature definition as a byte array that is expected to match with the file content, where `??` is used to match any byte. Finally, the `ep_only` line indicates whether the rule is expected to match only the bytes at the binary's entry point or anywhere in the file.

The following PEiD rule detects our running example from Fig. 1.

```
[2018-ISOLA-Tutorial -> PEiD]
signature = 49 20 61 6d 20 65 76 69 6c 21 21 21
ep_only = false
```

[1] https://www.aldeid.com/wiki/PEiD.

The signature matches the byte array that corresponds to the string "I am evil!!!", not limiting the match to the entry point (since `ep_only = false`). The command below shows that the string is present in the compiled binary for the malware.

```
$ hexdump -C malware_version1 | grep evil
000005e0  01 00 02 00 49 20 61 6d 20 65 76 69 6c 21 21 21  |....I am evil!!!|
```

```c
#include<stdio.h>
#include<wchar.h>

int malicious_behaviour(){
  wprintf(L"I␣am␣evil!!!\n");
}

int main(int argc, char **argv){
  malicious_behaviour();
}
```

Fig. 2. Our fake malware example, using wide characters to avoid simple string detection.

To easily bypass this detection the malware author can slightly change the malware to store the string using wide characters instead of ASCII characters (which does not change the malware's behavior). This modification is depicted in Fig. 2. In this case, looking for string "I am evil!!!" in the usual format will not work anymore. The command below shows that the string is detected only if specifically looking for wide characters.

```
$ hexdump -C malware_version2 | grep evil
$ strings -e L malware_version2 | grep evil
I am evil!!!
```

A new rule could be created to match wide-chars, however requiring a whole new rule for this illustrates one of the biggest limitations of PEiD rules: the lack of flexibility in the rule grammar. For instance, the most recent PEiD database[2] uses 829 rules (out of 3714) just to define version 1.25 of the VMProtect[3] packer.

2.3 Example Tool: DIE

Another tool to match pattern in files is DIE[4], which stands for "Detect It Easy". DIE supports a JavaScript-like scripting language for signatures. DIE allows the creation of flexible rules by using matching conditions, despite being limited by the lack of a well-defined code pattern for rule creation.

An example of a DIE rule is shown below.

[2] https://handlers.sans.org/jclausing/userdb.txt.
[3] http://vmpsoft.com/.
[4] https://github.com/horsicq/Detect-It-Easy.

```
// DIE's signature file

init("protector","PE␣Intro");

function detect(bShowType,bShowVersion,bShowOptions)
{
    if(PE.compareEP("8B04249C60E8........5D81ED........
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣80BD..........0F8548"))
    {
        sVersion="1.0";
        bDetected=1;
    }

    return result(bShowType,bShowVersion,bShowOptions);
}
```

The rule matches files protected with PE Intro, which is detected by an expected sequence of bytes at the entry point. The rule starts by declaring a new signature at "`init`" and then by proving a description of the rule in the "`detect`" function.

Like PEiD, DIE has a simple flag (`PE.compareEP`) determining whether to look for byte arrays at the entry point. DIE uses "`.`" as wildcards to match any byte. Rule matching is indicated by the variable `bDetected`, which is set to 1.

DIE also support more sophisticated rules that depends on multiple conditions and code reuse as in the rule below.

```
// DIE's signature file

includeScript("rar");

function detect(bShowType,bShowVersion,bShowOptions)
{
    detect_RAR(1,bShowOptions);
    return result(bShowType,bShowVersion,bShowOptions);
}
```

Among the main drawbacks of DIE are: its rule syntax, which is very verbose and requires an ad-hoc script for each rule, and also DIE's lack of documentation. Furthermore, DIE lacks features like annotations and well-defined modular interfaces.

2.4 Example Tool: YARA

A more modern tool and the current de facto standard is YARA [23]. YARA rules are defined using a JSON-like format and are meant to provide greater flexibility than PEiD and DIE rules. A YARA rule consist of strings (possibly including binary strings) and conditions that determine whether to trigger the rule. Furthermore, YARA rules provide annotations that simplify rule description, enable referencing between rules, and use modules that describe high-level file properties.

In our running example, both versions of the code in Fig. 1 and in Fig. 2 can be matched using a single YARA rule.

```
rule 2018ISOLATutorialYara {
  meta:
    description = "Example␣of␣YARA␣rule␣for␣ISOLA␣2018"
  strings:
    $ascii_string = "I␣am␣evil!!!"
    $wide_string  = "I␣am␣evil!!!" wide
  condition:
    $ascii_string or $wide_string
}
```

This example shows that the YARA rule format is more readable than PEiD's or DIE's and also that it can cover both wide and ASCII characters with a single rule. Still, the YARA grammar allows for even more straightforward description of the rule using multiple annotations for a single string.

```
rule 2018ISOLATutorialYaraSimpler {
  meta:
    description = "Simpler␣YARA␣rule␣for␣ISOLA␣2018"
  strings:
    $evil_string = "I␣am␣evil!!!" wide ascii
  condition:
    $evil_string
}
```

YARA provides many high-level modules allowing to include higher level properties in rules, like file size or entropy[5].

However, with small modifications to the code an adversary can bypass detection without changing any malicious behavior. One way to achieve this result is depicted in Fig. 3. In this version, the targeted string will not be contiguously placed in memory, therefore all the YARA rules above for the running example will fail to match.

```c
#include<stdio.h>

int malicious_behaviour(){
  printf("I␣am");
  printf("␣evil!!!\n");
}

int main(int argc, char **argv){
  malicious_behaviour();
}
```

Fig. 3. Our fake malware sample, with broken strings to avoid string-based detection.

These simple obfuscation techniques illustrate the limitations of syntactic signatures, showing a toy example of how malware can be modified to avoid syntactic pattern matching. It is easy for malware creators to create new versions of their malware that avoid syntactic pattern matching [20].

[5] http://yara.readthedocs.io/en/v3.5.0/modules.html.

2.5 Limitations: Obfuscating Against Syntactic Pattern Matching

In order to avoid pattern matching, it is common to use generic techniques like obfuscation in which important strings, parts of the code or even the whole code are transformed into some obfuscated representation. Generally, malware authors use obfuscation to hide information like the address of their Command & Control (C&C) server, targeted business, eventual credentials hardcoded into the sample, etc.

Figure 4 shows how our running example from Fig. 1 can be modified to remove any plain representation of the string targeted in our previous example. The commands below show that the string "I am evil!!!" is not contained in the file in either its ASCII or wide format.

```
$ strings malware_version4 | grep evil
$ strings -e L malware_version4 | grep evil
```

In this example, the string "I am evil!!!" is XORed with the keystream "ISOLA-TUTORIAL-2018" resulting in the following byte array `"0x00 0x73 0x2e 0x21 0x61 0x48 0x22 0x3c 0x38 0x6e 0x73 0x68 0x4b"`. The byte array is hardcoded along with the keystream so as to recover the original string whenever needed. Despite being an insecure practice to store the ciphertext along

```c
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

char *keystream = "ISOLA-TUTORIAL-2018";
char *obf = "\x00\x73\x2e\x21\x61\x48\x22\x3c\x38\x6e\x73\x68\x4b";

char *xor(char *str){
  int i;
  char *cipherstr;
  int len = strlen(keystream);
  cipherstr = malloc(len * sizeof(char));
  for(i = 0; i < len; i++) {
    cipherstr[i] = str[i] ^ keystream[i];
        if(cipherstr[i] == '\n') {
                cipherstr[i + 1] = '\0';
                break;
        }
  }
  return cipherstr;
}

int malicious_behaviour(){
  int i;
  //char *str = "I am evil!!!\n";
  char *str = xor(obf);
  printf("%s", str);
}

int main(int argc, char **argv){
  malicious_behaviour();
}
```

Fig. 4. Our fake malware sample, with XOR-obfuscated strings to avoid string-based detection.

with the key, the main intention of malware creators is to remain undetected until infection, rather than long term security.

The same idea can be achieved with other encryption methods. Nevertheless, one strategy employed to detect such cases is to pattern match constants defined in their algorithms, as shown below for a constant used by AES.

```
rule RijnDael_AES
{ meta:
    author = "_pusher_"
    description = "RijnDael␣AES"
    date = "2016-06"
  strings:
    $c0 = { A5 63 63 C6 84 7C 7C F8 }
  condition:
    $c0
}
```

Another possibility is to completely obfuscate the code by changing the syntactic structure of the binary. The commands below show how to compile the binary statically and how to pack it with UPX, and how this changes the syntactic properties of the binary without modifying its behavior.

```
$ gcc -static -o malware_version4 malware_version4.c
$ upx-3.94-amd64_linux/upx -f malware_version4 \
-omalware_version4_upx
$ readelf -h malware_version4 | tail -n 2
  Number of section headers:          33
  Section header string table index: 30
$ readelf -h malware_version4_upx | tail -n 2
  Number of section headers:          0
  Section header string table index: 0
$ ./malware_version4
I am evil!!!
$ ./malware_version4_upx
I am evil!!!
```

As we have seen, syntactic properties are easy to extract, however, since they are easily modifiable without changing the malicious behavior of the binary, they are also easy to bypass. This has a major impact on the effectiveness of syntactic signatures.

Hence, behavioral signatures have to be used to detect malware based on their behavior, since behavior is harder to automatically obfuscate. The next section explains how to extract behavioral signatures dynamically by executing the malware samples in a sandbox.

3 Dynamic Analysis: Sandbox Execution

Dynamic analysis refers to techniques that rely on executing a sample to analyze it. Sandbox execution lets the malware execute in an isolated environment, while tracking the malwareś behavior. Contrarily to the syntactic pattern matching methods presented in the previous section, this section builds a signature based on malware behavior which is resistant to syntactic obfuscation.

3.1 Principle

In order to provide an isolated environment, sandboxes typically rely on virtual machines (VMs) [10]. VMs exploit the fact that a processor can theoretically be simulated by a program. The Operating System (OS) running on a virtual machine is usually called the *Guest* while the OS running on the real hardware is usually called the *Host*.

In the context of malware detection, a sandbox isolates the effects of an untrusted binary to a VM, i.e. to the Guest OS without affecting the Host OS. In practice, some vulnerabilities in VMs [24] or in processors [33] may compromise such isolation. A snapshot of the state of the Guest is taken before each malware analysis, and the Guest is restored to this snapshot after the analysis.

The analysis of a binary in a sandbox relies on observations at various levels. Based on these observations, the binary is labeled or given a score which indicates whether it is likely to be malware. Typically, a sandbox observes the memory, interactions with the Guest OS, and network activity of the executed binary.

The memory is analyzed by dumping it to a file. This dump can be obtained by taking a snapshot of the VM during execution, which by design stores the full VM memory for resuming it. Tools such as Memoryze[6] and dumpit[7] are able to capture and save the full memory image of a physical machine or VM. The memory dump can then be analyzed a posteriori with dedicated tools such as Volatility [8,17]. Such analysis tools list the processes running, the opened ports, and the state of the windows registry at the time the memory was dumped. In particular, it is possible to retrieve artifacts such as uncompressed or unencrypted binaries that are temporary stored in the memory and can be analyzed further.

In order to observe the processes running in the sandbox, the binary can be launched with a debugger to observe all the steps done in the execution of the binary. Another option is to observe the execution by recording the system and library calls, along with their argument values. These calls track what the binary is actually doing in the system, since any action on the system (e.g. writing to a file, changing a registry key, sending a network packet, etc.) has to be done via such a call. Some techniques define various patterns of calls [2,4] (see [3] for a comparison of different kind of patterns) or rely on system call dependency graphs [14] to represent and recognize (malicious) behaviors. Such approaches often involve machine learning to classify the calls of an unknown binary based on patterns learned on known malware and cleanware. The software in charge of observing the processes (debugger and/or process monitor) needs to be running in the sandbox as well.

Finally, the network behavior of the malware can be observed from outside the sandbox, by looking at the traffic on the virtual network card of the sandbox. Also, the monitoring process in the sandbox can save the keys used for TLS traffic, in order to decrypt HTTPS packets. Depending on the context, the

[6] https://www.fireeye.com/services/freeware/memoryze.html.
[7] https://my.comae.io/login.

analyst can fake the Internet (i.e. reply with standard packets for each protocol) or monitor and block the traffic while allowing access to the real Internet. The latter approach is potentially more dangerous (i.e. a malware could potentially infect another system) but may enable some particular behavior that are not observable otherwise.

3.2 Example Tool: Cuckoo

Cuckoo[8] is an open-source sandbox written mainly in Python. In the sandbox, the Cuckoo agent handles the communication with the Host. The agent receives the binary to analyze as well as the analysis module, written in Python. The analysis module is in charge of performing the required observation from within the sandbox. Since the analysis module is uploaded to the sandbox along with the sample to execute, the agent can handle several types of analysis.

The default analysis module monitors and registers the system calls made by the binary to analyze and all its children processes. This information is then used to produce a score indicating whether the binary is malicious.

While analysis commands can be submitted from the command line, Cuckoo also features a web interface allowing the user to submit files to analyze and to receive the results of the analysis. We focus here on behavioral analysis since it is complementary to the syntactic analysis techniques presented in the previous section. Cuckoo includes YARA, which can also be used on the binary or its memory during the execution of the binary.

Time & API	Arguments	Status	Return	Repeated
brk July 3, 2018, 4:53 a.m.	p0: `0x0`		9397874 8002304	0
brk July 3, 2018, 4:53 a.m.	p0: `0x557921e63000`		9397874 8137472	0
write July 3, 2018, 4:53 a.m.	p2: `13` p0: `1` p1: `I am evil!!!`		13	0
exit_group July 3, 2018, 4:53 a.m.	p0: `0`			0

Fig. 5. End of the behavioral analysis report from Cuckoo for our malware sample in Fig. 3

Figure 5 shows the end of the call trace for the example from Fig. 3. In the trace, we can see that the argument passed to "write", which is the lower level call used for implementing "printf", is the string "I am evil !!!". This is sufficient to recognize the malicious behavior, even if syntactic signatures were unable to detect it. Of course, our malicious behavior is oversimplified here and is kind of trivial to recognize.

[8] https://cuckoosandbox.org/.

3.3 Limitations: Anti-Sandboxing Techniques

Some malicious binaries may try to attack the sandbox. A common approach is to try to detect that they are being run in a sandbox, and hide their malicious behavior in that case. If a sandbox is detected, the malware can try to crash it, like AntiCuckoo[9] does, or even try to infect it [33].

Most malware won't try to attack the Host of a sandbox because their goal is to remain hidden. A sample that would crash or take over the Host of a sandbox would indeed be immediately classified as highly suspicious. In fact, malware samples commonly delete themselves if they detect a sandbox. Therefore, we will focus here on detecting that the current execution is within a sandbox.

There are various techniques to detect that the current environment is a virtual environment. For instance, the default name of hardware devices in VirtualBox contains the string "VirtualBox", which can be easily detected. In a similar way, interrogating the features of the CPU via the x86 instruction `cpuid` can provide evidence of a virtual environment. Additionally, some malicious binaries analyze the documents and the activity of the user and don't execute if the number of documents is abnormally low. See [13] for more examples of sandbox detection and [31] for the sandbox detection techniques used by the recent Rakhni malware.

```c
#include <stdio.h>

int hv_bit(){
  int cpu_feats=0;
  __asm__ volatile("cpuid"
        : "=c"(cpu_feats) //output: ecx or rcx -> cpu_feats
        : "a"(1));        //input: 1 -> eax or rax
   return (cpu_feats >> 31) & 1;
}

void malicious_behaviour(){
  printf("I␣am");
  printf("␣evil!!!\n");
}

void benign_behaviour(){
  printf("I␣am␣nice.\n");
}

int main(){
  int i,a;
  a=hv_bit();
  if ( a > 0) {
    benign_behaviour();
  } else {
    malicious_behaviour();
  }
  return 0;
}
```

Fig. 6. Our fake malware sample, including sandbox detection.

[9] https://github.com/David-Reguera-Garcia-Dreg/anticuckoo.

For our running example, we provide in Fig. 6 a C program that relies on a low-level instruction to discriminate between a virtual environment and a non-virtual one. The cpuid argument provide an assembly-level way to interrogate the processor about its features, in order, for instance, to use optimized instructions if available. In particular, when the register eax is set to one before cpuid is called, it returns various information about the processor. Notably, the highest bit of the register ecx is set to 1 if the current execution is controlled by an hypervisor (i.e. a virtual machine). See [12] for the specification of the cpuid instruction.

The function hv_bit leverages this behavior to output that particular bit. The cryptic inline assembly code is specified using the GNU syntax: __asm__ volatile (<code> : <outputs> : <inputs>). The code is simply the cpuid instruction (line 6), the output specifies that the register c (here ecx or rcx) should be put in the variable named cpu_feats (line 7) and finally the input specifies that the register a (here eax or rax) is set to 1 (line 8). The returned value is the bit 31 of the cpu_feats variable (the binary and zeroes the bits that were originally at position 32–64 in the rcx register, if any). Finally, the main function executes the malicious behavior only if the hv_bit function returns 0 (we're not in a hypervisor).

Time & API	Arguments	Status	Return	Repeated
brk July 3, 2018, 5:59 a.m.	p0: 0x0		9401092 4871680	0
brk July 3, 2018, 5:59 a.m.	p0: 0x55809fca4000		9401092 5006848	0
write July 3, 2018, 5:59 a.m.	p2: 11 p0: 1 p1: I am nice.		11	0
exit_group July 3, 2018, 5:59 a.m.	p0: 0			0

Fig. 7. End of the behavioral analysis report from Cuckoo for our malware sample in Fig. 6

Figure 7 shows the output of Cuckoo when analyzing a binary obtained from the code in Fig. 6. As expected, the hypervisor bit is set to one and the output in this version is "I am nice." which corresponds to the benign behavior. However, running the same program on a non-virtual machine will print "I am evil !!!" showing that the malicious behavior executes in that case.

An expensive way to detect whether a sample is trying to evade a sandbox is to compare its behavior in different contexts [15], such as in a VM, in a hypervisor, on a bare metal machine (i.e. an isolated machine) and on an emulation platform. However, the various tests performed by modern malware such as Rakhni [31] would hide the malicious behavior in most of these contexts, based on the environment. For instance, Rakhni has a list of more that 150 names of tools used for process monitoring and analysis; if one of the running processes is in that list, Rakhni will hide its malicious behavior.

Note that these sandbox detection techniques succeed because dynamic analysis by sandboxing aims at executing only a single execution path of the malware binary analyzed. Hence, by prefixing the malicious behavior with multiple checks on the execution environments, malware creators can guarantee that the malicious behavior will be executed only if the system is not a sandbox, is used by a real user, does not have an antivirus software installed, and so on. To circumvent this protection, we need to follow these environment checks in the execution of the binary and ask, what does the malware do when the check succeeds and what does it do when the check fails? Only by exploring both possible execution paths we can arrive at analyzing the malicious behavior. This is the basic idea behind concolic analysis, described in the next section.

4 Concolic Analysis: Symbolic Execution

Concolic analysis does not execute the binary but rather simulates it, with the aim of covering as many of the execution paths as possible of the binary. This increases the probability of detecting malicious behavior that would not be executed in a sandbox due to sandbox detection techniques.

In concrete execution, variables are assigned with *concrete* values that are re-evaluated whenever some assignment statement is reached during the execution. In symbolic execution, variables are *symbolic*, i.e. are assigned with a set of constraints representing a set of possible concrete values.

In practice, as the execution of the program proceeds, symbolic variables accumulate constraints on the possible values of the concrete variables, whereas concrete variables keep only the last actual assigned value. Therefore, symbolic execution does not scale as well as symbolic execution.

In symbolic execution, conditional statements are evaluated not just as True or False as with concrete execution, but as *satisfiable* or *unsatisfiable* instead. This means that, for a given symbolic variable on a given conditional statement, if it is possible to satisfy both the conditional statement and its negation, the execution will take both paths, whereas in concrete execution only one of them would be taken.

As a result, while concrete execution is able to traverse only one trace of execution at a time, symbolic execution can traverse multiple traces of execution simultaneously. Thus, concolic execution aims to combine the efficiency and scalability of concrete execution with high code coverage of symbolic execution.

4.1 Principle

To illustrate the difference between concrete and symbolic execution, we will use the following toy example.

In this example, `x` is taken as a user input and then it is tested on being non-negative and a root for $x^2 - 3x - 4$. If `x` satisfies both conditions, then the

```c
1   #include<stdio.h>
2
3   /*
4           Trial and error to find a solution for x^2 - 3x - 4 = 0 for x >= 0
5   */
6   int main() {
7           int x;
8           printf("Let's␣try␣to␣solve␣x^2␣-␣3x␣-␣4␣=␣0␣for␣x␣>␣0.\n");
9           printf("Enter␣a␣value␣for␣x:␣");
10          scanf("%5d", &x);
11          if(x * x - 3 * x - 4 == 0) {
12                  if(x >= 0)
13                          printf("%d␣is␣a␣positive␣root.\n", x);
14                  else
15                          printf("x␣has␣to␣be␣positive.\n");
16          }
17          else
18                  printf("%d␣is␣not␣a␣root.\n", x);
19  }
```

Fig. 8. Sample code for a trial and error root solution.

`printf` at line 12 is reached and we say that the execution *succeeded*, otherwise the execution *failed*.

This example's concrete execution is straightforward: a value is assigned to `x`, then it is checked for being a root and non-negative. It is very efficient to test whether the input satisfies both conditions or not, however it is not equally easy to find one value for `x` in which the execution succeeds.

Using concrete execution it is possible to find an `x` for which the execution succeeds randomly taking values for `x` and executing the program. However, this approach might succeed with a very low probability, and when there is no root for the equation, this approach will run end forever. Two traces of execution (for `x` $= -1$ and `x` $= 0$) are displayed in Fig. 9.

In contrast, it is possible to use symbolic execution to explore traces of execution in a more structured way. Using symbolic execution, `x` is assigned to a symbolic variable which will accumulate constraints along different execution paths. The equality for $x^2 - 3x - 4 = 0$ is met when `x` $= -1$ or `x` $= 4$, therefore it follows:

- Line 11 will be reached only if `x` is constrained to one of the root values (i.e. -1 or 4). For any other case, the execution will reach line 17.
- Line 12 is reached only if `x` is constrained to root values (line 10) and if it is positive (line 11), otherwise line 14 is reached.

Hence the execution *succeeds* and line 12 is reached only if `x` $= 4$. Figure 10 depicts how symbolic execution proceeds with the constraints of each path and is able to build a tree of execution traces, extending the single execution trace explored by concrete execution.

4.2 Example Tool: angr

angr [27] is a tool enabling concolic execution of binaries, written in Python and composed of different modules.

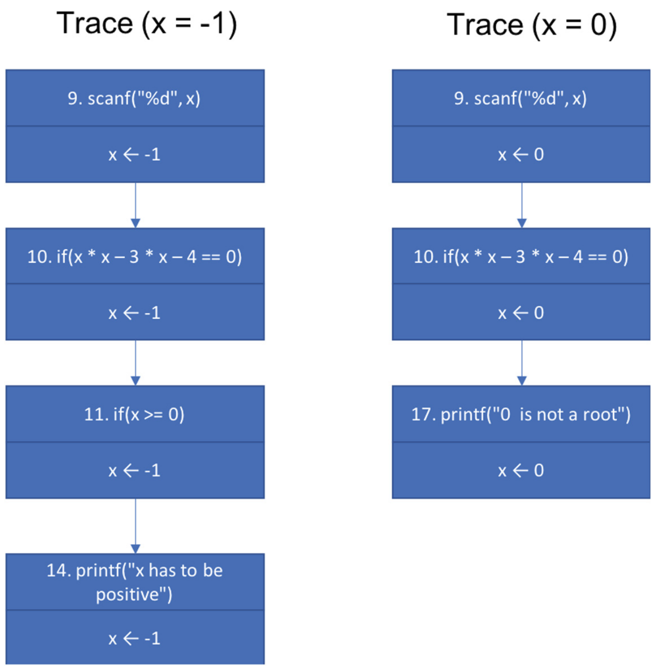

Fig. 9. Example of concrete traces of trial and error root solution.

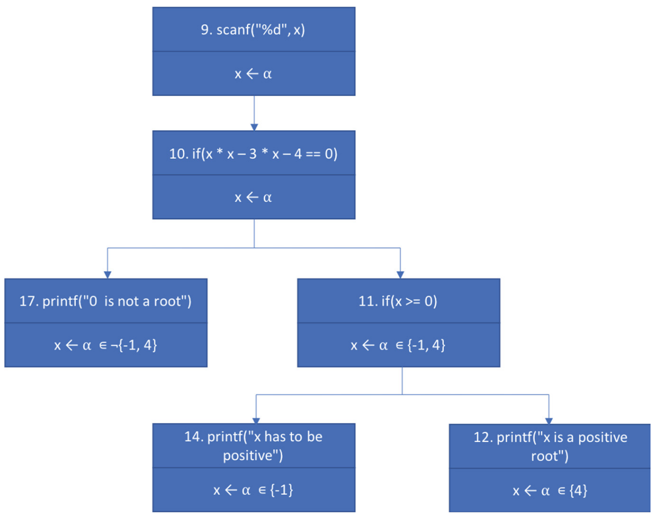

Fig. 10. Example of symbolic tracing of trial and error root solution.

– CLE: stands for "CLE Loads Everything" and is responsible for loading binaries and libraries.
– Archinfo: contains architecture-specific information.
– PyVEX: Python module to handle VEX, which is an intermediate representation that enables angr to work on different architectures.
– Claripy: module that interfaces with a constraint solver.

The execution starts by loading the binary, using the CLE module. To do so, CLE needs information about the architecture the program is target for, which is provided by Archinfo.

Once the binary is loaded, the angr symbolic execution engine coordinates the symbolic execution. The analysis unit of angr's analysis is the *basic block*, defined as a continuous sequence of code that has no branching (such as jumps or calls), and for each basic block angr creates a new state. A state contains the program's memory, registers, file system, and any other so-called "live data".

angr's execution evolves in steps. Each step transforms the current active states into their successor states. Constraints accumulated from past basic blocks/states are *solved* whenever some instruction depends on a symbolic variable (e.g. memory address). If the current state ends up in a conditional jump, angr evaluates the condition against the current constraints and proceed as follows.

– If both the conditional and its negation are satisfiable, angr creates two new successors states, one for each of the two possible states.
– If only one of the conditional and its negation is satisfiable, angr creates only one new successor state.
– If neither the conditional nor its negation are satisfiable, angr marks the state as deadended and terminates its execution.

Constraints are solved by SMT solvers interfaced by Claripy. Currently, the default SMT solver used is Microsoft's Z3, however others can be plugged into angr by writing an appropriate Claripy backend.

The procedure above is able to emulate a bare metal environment, including abstractions provided by the underlying operating system, such as files, network, processes and others. In order to fulfill these abstractions, angr includes a module called SimOS, which provides all the required OS objects during the analysis.

Finally, to allow angr to work with multiple architectures, instead of running concolic analysis on instructions charged by CLE, these instructions are first lifted to the VEX intermediary representation before the analysis is done. Multiple architectures can be lifted to VEX without any loss in the overall analysis.

Figure 11 from [29] depicts the relationship between the different modules.

To illustrate the benefits of concolic analysis, we use the code example in Fig. 8. Running a full automated analysis with angr, it is possible to verify that angr reaches 3 final states (i.e. deadended states), corresponding to the leaves of the execution tree presented in Fig. 10.

Figure 12 shows the commands required to perform the analysis. It is possible to check that for each one of the three deadended states, the analysis reached a different line of the original code (corresponding to a different printed message).

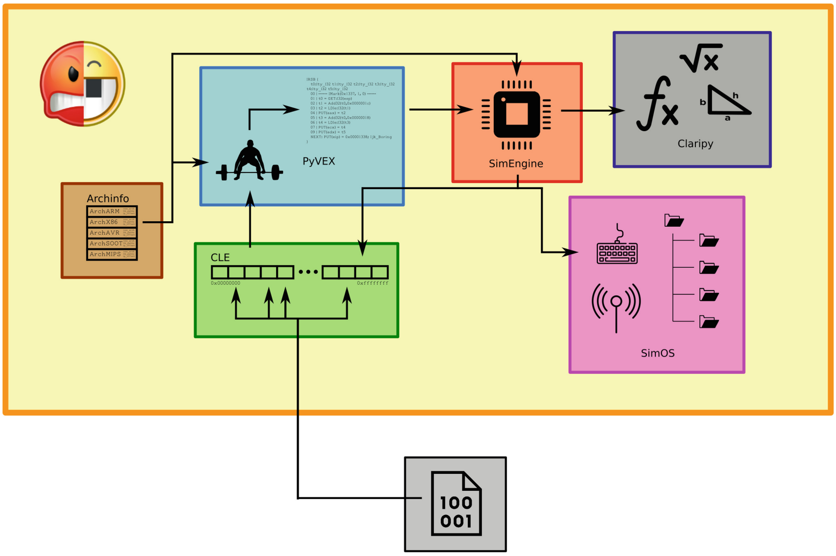

Fig. 11. Relationship between angr modules.

```
In [1]: import angr
WARNING | 2018-07-04 16:19:17,147 | angr.analyses.disassembly_utils | Your verison of capstone does not support MIPS instruction groups.

In [2]: p = angr.Project("root")

In [3]: state = p.factory.entry_state()

In [4]: sm = p.factory.simgr(state)

In [5]: sm.run()
WARNING | 2018-07-04 16:20:04,490 | angr.manager | No completion state defined for SimulationManager; stepping until all states deadend
WARNING | 2018-07-04 16:20:05,052 | angr.state_plugins.symbolic_memory | Concretizing symbolic length. Much sad; think about implementing.
Out[5]: <SimulationManager with 3 deadended>

In [6]: s0 = sm.deadended[0]

In [7]: s1 = sm.deadended[1]

In [8]: s2 = sm.deadended[2]

In [9]: s0.posix.dumps(0)
Out[9]: '42222'

In [10]: s0.posix.dumps(1)
Out[10]: "Let's try to solve x^2 - 3x - 4 = 0 for x > 0.\nEnter a value for x: 42222 is not a root.\n"

In [11]: s1.posix.dumps(0)
Out[11]: '-0001'

In [12]: s1.posix.dumps(1)
Out[12]: "Let's try to solve x^2 - 3x - 4 = 0 for x > 0.\nEnter a value for x: x has to be positive.\n"

In [13]: s2.posix.dumps(0)
Out[13]: '00004'

In [14]: s2.posix.dumps(1)
Out[14]: "Let's try to solve x^2 - 3x - 4 = 0 for x > 0.\nEnter a value for x: 4 is a positive root.\n"
```

Fig. 12. Example of angr analysis on trial and error root solution.

For a given state, `stdin` can be accessed through *posix.dump(0)* while `stdout` can be accessed by *posix.dump(1)*. Hence it is straightforward to verify that the execution succeeds if $x = 4$.

4.3 Limitations: Symbolic Explosion

The ability to explore all the possible traces of a given binary is one of the core advantages of concolic execution. However, this can be exploited by malware authors to generate a binary for which concolic execution has a huge number of possible paths. For instance, starting the binary by a sequence of n conditional jumps depending on a value that is not treated as concrete by the symbolic execution will possibly create 2^n traces. The malware author can use this technique to try to fill the memory of the machine running the symbolic execution. A way to mitigate that technique is not to a breadth-first search (BFS) for ordering the exploration of the reachable states.

Another weakness of symbolic execution is the constraint solver. Indeed, some malware include *opaque predicates* that are commonly used to hinder static analysis [19]. A simple example is to write a complicated conditional jump whose condition always evaluates to true, so that at runtime only one branch of the conditional jump is taken, but static analysis requires a lot of effort to conclude that the predicate is always true. In the context of symbolic execution, such predicates might become very complex expressions involving several symbolic variables. These predicates will be analyzed by the constraint solver, forcing the solver to use a large amount of resources (time and memory) to try to solve the constraint. If the constraint solver runs out of resources the concolic execution has to analyze both paths, and complex dead code (possibly including more opaque predicates) can be inserted in the path that is never selected at runtime to hinder the analysis.

Finally, symbolic execution also struggles with just-in-time (JIT) code, i.e. code that writes the next instructions to execute in the memory just before executing them. If some of the written instructions are symbolic at the time of writing, the execution now faces the problem of having a symbolic instruction to execute. One expensive solution would be to list all instructions that meet the constraint of the symbolic instruction and try each of them. Otherwise the symbolic execution can be stopped at this point or miss some branches [34]. A similar problem is encountered whenever the address of a jump is symbolic.

For our running example, we use Tigress [7] to produce an obfuscated version of the code from Fig. 6. The commands presented in Fig. 13 yields a *a.out* executable that is obfuscated against symbolic execution. When trying to analyzing the executable with angr, the tool will execute without terminating.

```
echo "#include␣\"$TIGRESS_HOME/jitter-amd64.c\"" > tmp.c
cat malware_version5.c >> tmp.c
tigress --out=malware_version5.c  \
  --Environment=x86_64:Linux:Gcc:4.6 \
  --Transform=Jit --Functions=main tmp.c
```

Fig. 13. Command using Tigress to builds a version of the code of Fig. 6 using JIT compilation to obfuscate the code, to prevent angr from analyzing it.

5 Conclusion

This paper presents different techniques to perform malware detection based on different kinds of signatures: syntactic signatures by static analysis; and behavioral signatures by dynamic and concolic analysis. Each technique is presented with a simple example, and with an example of how to counter it based on its limitations.

The techniques are presented in increasing order of complexity, and justified by the fact that each one is effective against the techniques hindering the previous one.

– Syntactic pattern matching is hindered by packing and obfuscation.
– Packed and obfuscated malware can be analyzed by dynamic execution in a sandbox.
– Sandboxes can be prevented from observing malicious behavior by sandbox detection.
– Sandbox detection can be countered by concolic execution.
– Concolic execution can be prevented from finding interesting execution branches by opaque predicates and JIT compilation.

However, in practice the different costs of these techniques mean that even simple ones like string-based static detection should not be discarded. YARA is widely used by security researchers, and in fact it is common to complement reports on new malware with YARA rules able to detect such malware, exactly because YARA is an efficient and optimized tool for pattern matching whose cost is negligible compared to the cost of starting a sandbox or a concolic binary execution engine.

Hence, all of the tools and techniques presented are useful to security analysts in different scenarios. Binaries that are detected as suspicious but not definitively malicious by static analysis, for instance because they employ packing and other obfuscation techniques, can be analyzed in a sandbox to characterize their behavior. Were the sandbox to fail due to sandbox evasion, the analyst can employ concolic analysis, and so on. Advanced malware can require multiple tools and significant analysis time by an expert before it is thoroughly dissected and understood, but this can lead to the creation of syntactic and behavioral rules to automatically detect new samples of the malware in the future without having to repeat the analysis.

References

1. Agrawal, H., Bahler, L., Micallef, J., Snyder, S., Virodov, A.: Detection of global, metamorphic malware variants using control and data flow analysis. In: 31st IEEE Military Communications Conference, MILCOM 2012, Orlando, October 29 – November 1, 2012, pp. 1–6 (2012). https://doi.org/10.1109/MILCOM.2012.6415581
2. Alazab, M., Venkatraman, S., Watters, P., Alazab, M.: Zero-day malware detection based on supervised learning algorithms of API call signatures. In: Proceedings of the Ninth Australasian Data Mining Conference, vol. 121, pp. 171–182. AusDM 2011, Australian Computer Society Inc., Darlinghurst (2011). http://dl.acm.org/citation.cfm?id=2483628.2483648

3. Canali, D., Lanzi, A., Balzarotti, D., Kruegel, C., Christodorescu, M., Kirda, E.: A quantitative study of accuracy in system call-based malware detection. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis, pp. 122–132. ISSTA 2012. ACM, New York (2012). https://doi.org/10.1145/2338965.2336768
4. Christodorescu, M., Jha, S., Seshia, S.A., Song, D., Bryant, R.E.: Semantics-aware malware detection. In: 2005 IEEE Symposium on Security and Privacy (SP 2005), pp. 32–46, May 2005. https://doi.org/10.1109/SP.2005.20
5. Cisco: Annual Cybersecurity Report (2018). https://www.cisco.com/c/m/en_au/products/security/offers/cybersecurity-reports.html
6. ClamAV: Clamav 0.99b meets yara! ClamAV blog. https://blog.clamav.net/2015/06/clamav-099b-meets-yara.html
7. Collberg, C., Martin, S., Myers, J., Nagra, J.: Distributed application tamper detection via continuous software updates. In: Proceedings of the 28th Annual Computer Security Applications Conference. ACSAC 2012, pp. 319–328. ACM, New York (2012). https://doi.org/10.1145/2420950.2420997
8. Ehsan, F.: Detecting unknown malware: security analytics & memory forensics. Presentation at RSA 2015 Conference (2015). https://www.rsaconference.com/events/us15/agenda/sessions/1517/detecting-unknown-malware-security-analytics-memory
9. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 213–223. PLDI 2005, ACM, New York (2005). https://doi.org/10.1145/1065010.1065036
10. Goldberg, R.P.: Survey of virtual machine research. Computer **7**(6), 34–45 (1974)
11. Idika, N.C., Mathur, A.P.: A survey of malware detection techniques (2007)
12. Intel: Intel® 64 and ia-32 architectures software developer's manual combined volumes 2a, 2b, 2c, and 2d: Instruction set reference, a-z. Technical report, May 2018. https://software.intel.com/sites/default/files/managed/a4/60/325383-sdm-vol-2abcd.pdf, order Number: 325383–067US
13. Jung, P.: Bypassing sanboxes for fun! Presentation at hack.lu (2014). http://archive.hack.lu/2014/Bypasss_sandboxes_for_fun.pdf
14. Karbalaie, F., Sami, A., Ahmadi, M.: Semantic malware detection by deploying graph mining. Int. J. Comput. Sci. Issues (IJCSI) **9**(1), 373 (2012)
15. Kirat, D., Vigna, G., Kruegel, C.: Barecloud: bare-metal analysis-based evasive malware detection. In: USENIX Security Symposium, pp. 287–301 (2014)
16. Kuzurin, N., Shokurov, A., Varnovsky, N., Zakharov, V.: On the concept of software obfuscation in computer security. In: Garay, J.A., Lenstra, A.K., Mambo, M., Peralta, R. (eds.) ISC 2007. LNCS, vol. 4779, pp. 281–298. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-75496-1_19
17. Ligh, M.H., Case, A., Levy, J., Walters, A.: The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory. Wiley, Indianapolis (2014)
18. MissMalware: Tdsanomalpe - identifying compile time manipulation in pe headers. Miss Malware blog. http://missmalware.com/2017/02/tdsanomalpe-identifying-compile-time-manipulation-in-pe-headers/
19. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007), pp. 421–430, December 2007. https://doi.org/10.1109/ACSAC.2007.21

20. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: ACSAC, pp. 421–430. IEEE Computer Society (2007). http://dblp.uni-trier.de/db/conf/acsac/acsac2007.html#MoserKK07
21. Pietrek, M.: Peering inside the PE: A tour of the win32 portable executable file format. Microsoft Developer Network blog (1994). https://msdn.microsoft.com/en-us/library/ms809762.aspx
22. Preda, M.D., Christodorescu, M., Jha, S., Debray, S.: A semantics-based approach to malware detection. SIGPLAN Not **42**(1), 377–388 (2007). https://doi.org/10.1145/1190215.1190270, https://doi.org/10.1145/1190215.1190270
23. Project, Y.: Yara documentation. https://yara.readthedocs.io/
24. Schwartz, M.: Oracle virtualbox multiple guest to host escape vulnerabilities. SecuriTeam Secure Disclosure blog (2018). https://blogs.securiteam.com/index.php/archives/3649
25. Sen, K., Marinov, D., Agha, G.: Cute: a concolic unit testing engine for c. In: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 263–272. ESEC/FSE-13, ACM, New York, (2005). https://doi.org/10.1145/1081706.1081750
26. Sharma, A., Sahay, S.K.: Evolution and detection of polymorphic and metamorphic malwares: a survey. Int. J. Comput. Appl. **90**(2), 7–11 (2014)
27. Shoshitaishvili, Y., et al.: SoK: (state of) the art of war: offensive techniques in binary analysis. In: IEEE Symposium on Security and Privacy (2016)
28. Sikorski, M., Honig, A.: Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software, 1st edn. No Starch Press, San Francisco (2012)
29. Subwire, l.: throwing a tantrum, part 1: angr internals. Angr blog. http://angr.io/blog/throwing_a_tantrum_part_1
30. Szor, P.: The Art of Computer Virus Research and Defense. Addison-Wesley Professional, Boston (2005)
31. Vasilenko, E., Mamedov, O.: To crypt, or to mine - that is the question. post on Securelist - Kaspersky Lab's cyberthreat research and reports (2018). https://securelist.com/to-crypt-or-to-mine-that-is-the-question/86307/
32. VirusTotal: Malware hunting. https://www.virustotal.com/#/hunting-overview
33. Wojtczuk, R., Rutkowska, J.: Following the White Rabbit: Software attacks against Intel (R) VT-d technology (2011). http://www.invisiblethingslab.com/resources/2011/Software%20Attacks%20on%20Intel%20VT-d.pdf
34. Yadegari, B., Debray, S.: Symbolic execution of obfuscated code. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 732–744. ACM (2015)