SAPIENZA
UNIVERSITÀ DI ROMA

Symbolic Execution of Malicious Software: Countering Sandbox Evasion Techniques

Faculty of Information Engineering, Informatics and Statistics

Master of Science in Engineering in Computer Science

Candidate

Fabio Rosato

ID number 1565173

Thesis Advisor

Prof. Camil Demetrescu

Co-Advisor

Dr. Daniele Cono D'Elia

Academic Year 2016/2017

Symbolic Execution of Malicious Software: Countering Sandbox Evasion Techniques
Master thesis. Sapienza – University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Version: October 14, 2017

Author's email: rosato.1565173@studenti.uniroma1.it

Mater artium necessitas

- Latin proverb

Contents

Introduction

Malicious software, commonly known as malware, is an ever growing threat. The number of new malware samples and the corresponding economic damage caused by them increase year by year. Malware detection and analysis has always been and continues to be the first and most important line of defense.

Analysis techniques, generally categorized as either dynamic or static depending on whether the program is actually executed or not during the analysis, are constantly evolving and improving to better face their offensive counterparts. The complexity of the code obfuscation measures widely employed by malware, along with the trend towards automation justified by the rising amount of new samples surfacing every day, have largely favored dynamic analysis techniques over static ones. However, since DARPA announced in 2013 a Cyber Grand Challenge for the development of automatic defense systems able to discover and correct software flaws in near real-time, there has been a growing interest in the applicability of symbolic execution techniques to the security domain.

Symbolic evaluation, while technically being a static program analysis technique (since the code is not actually run on the machine), effectively blurs the line between the two approaches. In a symbolic execution an interpreter follows the program code, assuming symbolic values for inputs rather than obtaining concrete ones. The interpreter is then potentially able to explore all reachable states in the control flow graph of the program, forking new paths at each branching point involving a symbolic value. Through the aid of a constraint solver, the symbolic engine can determine which branches are feasible (i.e., they can actually be taken in a concrete execution). Additionally, symbolic inputs can be resolved to actual concrete values that satisfy the corresponding path constraints when needed.

While symbolic execution is not a novel idea (it was first introduced in the 1970s), its applications for many years have been practically exclusively confined to general software testing, though with excellent results. Recently, however, its applicability to the security domain, and in particular to malware analysis, has

been investigated by a growing number of academic works. Its ability to potentially cover all possible execution paths and subsequently identify the corresponding concrete input values that would elicit them, make symbolic evaluation an ideal instrument for the study of the trigger-based behaviors extremely common in malware.

Unfortunately, there is no such thing as a free lunch. A fundamental problem of symbolic execution is poor scalability due to *path explosion*. Each branching point involving a symbolic value doubles the number of paths, making this number effectively exponential with respect to the size of the executable. The many heuristics and optimizations that have been researched and studied over the years in order to mitigate the path explosion problem, predictably target general software testing, focusing on metrics like code coverage. The application of symbolic execution to malware analysis is then complicated, aside from the numerous obfuscation and anti-analysis techniques typical of the malware domain, by the widespread adoption of evasive measures. The vast majority of malware evasive techniques reactively try to detect the presence of an hostile environment, be it a security product or an analysis-related application. Detection techniques are inherently based on branching and thus hit hard the fundamental weakness of symbolic execution. What aggravates the situation is that evasive checks represent critical branching points: if the check detects the presence of an hostile entity, the malware generally dissimulates its behavior in order to appear innocuous to the analysis environment. This means that at least half of the paths generated by a single evasive check do not lead to any relevant malicious behavior and are thus practically useless from an analysis standpoint. It is then fundamental, for any conceivable application of symbolic execution to the malware domain, to solve this problem.

In this thesis we perform a meticulous profiling of the most widely employed malware evasion techniques, proposing first a precise high-level categorization, and then following up with a detailed study of the behavioral patterns of the most common detection techniques, especially in terms of their interaction with the Windows operating system APIs. Along with this study, we developed an extension of the Angr open source binary analysis and symbolic execution framework, that is able to constrain the symbolic exploration in order to automatically pass the evasive detection checks. Such a solution is indeed ideal, we argue, since it lets us effectively bypass the symbolic exploration of the evasive check, completely avoiding the useless path overhead introduced by the corresponding branching points. The

developed extension was tested both on a synthetic evasion demonstration tool and on real evasive malware, and proved to be both very efficient and easily adaptable when necessary.

Thesis structure The rest of this thesis is structured as follows.
In Chapter 1, after a brief overview of malware detection and analysis, we define malware evasion and introduce our high-level categorization of the techniques, concluding with a brief survey of the most widely adopted anti-evasion measures.
Chapter 2 introduces symbolic execution, detailing both its strengths and weaknesses, before presenting ANGR, the symbolic execution engine we used for developing and testing our ideas.
In Chapter 3, after explaining exactly why malware evasion represents a severe hindrance to the applicability of symbolic execution, we give a detailed account of the most common detection techniques, presenting for each check the patch we developed in our ANGR extension to bypass its exploration.
A detailed account of the evaluation of our extension and the proposed patches is then provided in Chapter 4, first on a synthetic demonstration tool (*pafish*) and then on a real malware sample.
Finally, in Chapter 5 we present our conclusions and suggest possible future developments.

Chapter 1

Malware evasion

In this chapter we will give the reader an overview of evasive techniques employed by malware in order to thwart analysis environments. We will first explain what exactly is malware evasion and why it matters. Then we will give a general overview and categorization of the existing evasive techniques. Finally we'll conclude with a brief overview of the currently available and employed anti-evasion techniques, that is the techniques used in malware analysis to thwart evasive attempts in the never-ending game of cat and mouse played by malware analysts and malware authors.

1.1 Malware analysis and malware evasion

Malware, short for "malicious software", is an umbrella term used to refer to any form of harmful, intrusive, or otherwise hostile software. Malware is defined by its malicious intent, acting against the requirements of the user. According to AV-Test (one of the most renowned test institutes for anti-malware products) more than 250,000 *new* malware samples are registered every day [15]. The word new deserves stressing, since the count is not based on files: these samples represent new specimen, with code properties not fingerprinted before. Additionally, all signs point to a continued growth, with significant increases over the years.

Detection and analysis are two clear fundamental objectives in the war against malicious software. Once a malware is detected and analyzed, the usual threat response takes place, and the malware damage potential is significantly reduced. Not surprisingly then, one of the fundamental goals of practically all malicious software is to avoid detection. And to hinder detection and analysis, malware authors have employed over the time an incredibly large and ever growing set of

tricks and techniques.

However, the definition of malware evasion we employ in this thesis is more specific and requires a little understanding of the various approaches to malware analysis currently used in the security industry, beginning with the distinction between static and dynamic malware analysis.

1.1.1 Static malware analysis

Static malware analysis (also known as code analysis) refers to any type of analysis that is conducted *without executing the sample*. It is usually performed dissecting the different resources of the sample binary file and studying each component individually in order to determine the functionality of the program.

Typical activities performed in static analysis are

- analysis of strings
- analysis of imported functions
- anti-virus fingerprinting
- analysis of disassembled code.

Malware authors hinder static analysis employing several obfuscation techniques [23], primarily *packing* [20]. Briefly, packing refers to the process of compressing, encrypting, or otherwise transforming an executable file, producing a new, *packed* executable that is semantically equivalent to the original one (i.e. it has the same functionality). While this process is sometimes employed in general non-malicious software (especially in order to reduce the size of the distributed file), it is literally abused by malware authors in order to harden a sample: the packed executable is a lot harder to analyze, making all of the previously mentioned activities practically impossible using static approaches alone.

1.1.2 Dynamic malware analysis

Dynamic malware analysis (also known as behavioral analysis) is performed observing the behavior of the malware while it is actually *running* on a host system, i.e. by detonating the sample in a controlled analysis environment and studying its actions [22]. A controlled analysis environment, also known as a ***sandbox***, is fundamental in order to contain potentially malicious applications and study them safely. Many sandboxes are actually implemented as *virtualized systems* that can be easily rolled back to a clean state once the analysis is

complete.

Typical activities performed in dynamic analysis are

- debugger-assisted execution
- process monitoring
- file system monitoring
- network traffic inspection.

While most static analysis techniques generally require a time consuming manual approach that can't possibly scale with the incredibly large amount of samples surfacing every day, dynamic analysis techniques are usually automated and very fast. The usual modus operandi in the security business is then to sift through the horde of samples surfacing everyday employing automated tools, reserving the professional touch of human analysts for only the most threatening of samples. Automated dynamic analysis environments thus represent the first and most important line of defense against malware threats. Evading automated analysis products is then one of the fundamental objectives of malware, and exactly what so-called **evasive techniques** aim to do.

1.1.3 Malware evasion defined

After this brief but instrumental overview of malware analysis, we can finally define **malware evasion** as the set of techniques employed by malware to avoid being detected *by an automated analysis product*. An evasive malware is a malware that exhibits no malicious behavior in a sandbox, but that infects the intended target.

Malware evasion over the years has gone from niche to mainstream. Individual malware samples are including more and more evasive behaviors. While a couple of years ago only a small fraction of malware showed any signs of evasion, today a sizeable portion (more than 80%) is actually evasive. And while evasive malware a few years ago tended to leverage at most two or three evasive tricks, much of today's evasive malware is tailored to bypass detection using as many as 10 or more different techniques [19].

Malware evasion has become so common that it is actually commoditized and automated by all the major malware packers and crypters commonly available on the web.

1.2 Malware evasion categorization

Evasive techniques are generally superficially categorized according to what particular activity or product they try to thwart. So we may read about *anti-debugging* techniques, *anti-virtualization* techniques, and *anti-sandbox* techniques.

In this thesis however we will present an alternative, more detailed categorization of evasive techniques, focusing more on the behavioral pattern of the technique rather than its objective. This categorization will prove useful when studying concrete instances of a technique, and in particular when devising anti-evasion patches in the later chapters.

A first clear distinction can be made regarding the **general approach** of the technique: evasive techniques either try to explicitly detect the presence of an *hostile entity*, so that the malware can *dissimulate* its malicious behavior (i.e. a **reactive**, defensive approach), or they unconditionally perform some actions that are *damaging* for that particular entity, but innocuous otherwise (i.e. they use a **proactive**, offensive approach) [1].

Generally speaking, all techniques falling under the *reactive-detection* branch try to detect an hostile environment exploiting any *peculiarity* it may have compared with a normal environment. Under this branch we further classify according to the particular **clue** analyzed for the detection:

- **Artifact** (or software-based) detection, where an artifact is any software by-product created by the analysis environment. This category contains the vast majority of the most common techniques effectively employed by malware. It can be further subdivided in
  - *OS artifacts*, i.e. artifacts produced in the operating system of the machine hosting the environment, e.g. Interrupt Descriptor Table relocation or presence of analysis processes running.
  - *Process artifacts*, i.e. artifacts related to the process that is hosting the executable (i.e. the executing instance of the program), e.g. injected DLLs and hooked APIs.
  - *File system artifacts*, i.e. artifacts created in the file system of the hosting machine, e.g. executables, drivers, and DLLs.

[1] An interesting alternative is the reactive, offensive approach, i.e. the malware only damages the intended victim. This approach is very evasive if the intended victim is rare and specific. Not surprisingly then, general malware practically never does this, given that one of its main objectives is spread maximization.

- **Hardware-based** detection, denoting any detection technique that exploits hardware differences between an analysis environment and a normal one. Because of resource sharing, analysis environments typically have very modest hardware, e.g. small amount of volatile memory, little disk size, or single-core CPUs.
- **Time-based** detection, indicating any technique that tries to detect an analysis environment exploiting timing discrepancies with a normal environment. These differences are practically always present, either due to overhead or explicit modifications and patching.
- **Human user** detection (i.e. *reverse Turing tests*), describing any technique that tries to detect an automated analysis environment by checking for the presence (or rather the absence) of a human user, e.g. mouse movement, key pressing, or file system *wear-and-tear*.

On the *proactive-offensive* branch instead we may further classify according to the **attack type**:

- **Time-based** attacks: because of the resource sharing required by the large amount of samples to be analyzed, automated analysis environments typically run a sample for a very limited time span, generally in the order of a couple of minutes. Malware then may try to play for time, e.g. *stalling*, *sleeping*, or otherwise *delaying* the execution of the malicious payload. Many analysis environments, as a result, patch sleeping/waiting functions abused by malware. In response, malicious programs have been known to employ race conditions that would prove harmless in a normal environment, but crash the program in a patched one.
- **CPU instruction-based** attacks: these attacks exploit special machine instructions to which the analysis product is somehow sensitive, e.g. abusing the `INT 3` instruction that is used by debuggers to set software breakpoints, thus disrupting the debugging process.

In this thesis, however, we will be focusing exclusively on the *reactive-detection* branch, since it is the inherent branching nature of those techniques that represents a problem for the application of symbolic execution to the analysis. Proactive techniques are unconditional, they cause practically no forking in the control flow, and as such they don't generally need any special attention when executed symbolically.

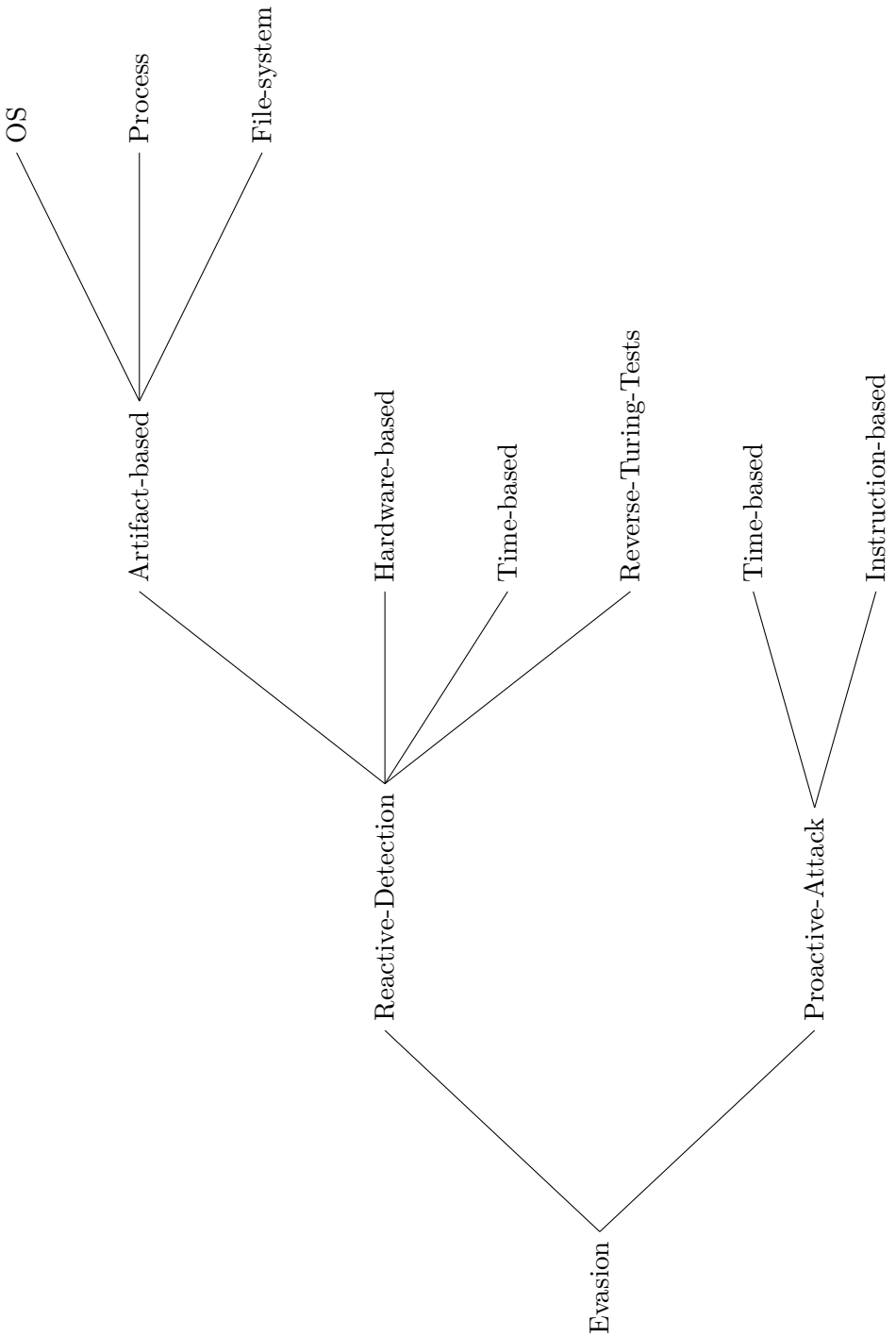


Figure 1.1. Evasive techniques categorization

1.3 Anti-evasion

Automated analysis environments, on their part, try to counteract evasive techniques using various measures. As is often the case in malware analysis, defensive measures are a lot more complex than the attacks they try to prevent, and anti-evasion is no exception. Anti-evasion techniques generally fall under two possible branches, mirroring the same general approaches to evasion:

- **reactive** approach, i.e. detect evasion and try to mitigate
- **proactive** approach, i.e. prevent evasion in the first place.

A conceptually simple instance of reactive approaches to anti-evasion is for example artifact monitoring, in which the analysis product monitors access to artifacts generally checked by evasive techniques. More complicated examples of reactive approaches employ more involved forms of program profiling, for example measuring *progress* made by the executable (particularly useful for stalling loops), or code coverage. Both of these examples effectively turn the tables on evasive malware, using the evasion techniques they employ to avoid detection as a way to detect them in the first place. Unfortunately, reactive approaches generally tend to be very brittle, and, most importantly, each additional intervention usually ends up producing new ways for a malware to fingerprint and detect the system, thus making the whole thing an endless cat-and-mouse game.

Proactive approaches, on the other hand, try to prevent detection in the first place, generally making the analysis product stealthier, i.e. more transparent to the analyzed samples. Straightforward instances of proactive approaches consist in minimizing the analysis product fingerprint, for example removing detectable artifacts or randomizing their values. More involved examples are the so-called *bare-metal* and *hardware-assisted* solutions.

Bare-metal systems run the sample on real hardware rather than a virtualized environment. They are clearly effective against anti-virtualization techniques, but generally lose all the benefits of running on a virtual system, namely inexpensiveness and fast-rollback to a clean state. However, various approaches have been proposed to improve their scalability [4, 6]

Hardware-assisted solutions, use hardware features to increase the transparency of analysis processes (e.g. debugging) running on bare-metal systems [8]. These approaches generally exploit alternative processor operating modes (e.g. ring-2, also known as System Management Mode, or SMM for short), in order to run in an isolated processor environment and operate transparently to both the OS and

general applications.

Finally, many hybrid approaches have been proposed [3, 7] - and are probably secretively employed in the security business - that run samples both in a virtualized and a bare-metal system *if necessary*. The whole area is a quite active research branch.

Chapter 2

Symbolic Execution

In this chapter we will give the reader an overview of symbolic execution, concisely detailing both its strengths and weaknesses. In the end we will also introduce Angr, the open source symbolic execution engine we used for developing and testing our ideas.

2.1 Overview

In computer science, **symbolic execution** (also known as **symbolic evaluation**) is a popular program analysis technique introduced in the mid '70s. It is generally used to determine what inputs cause each part of a program to execute, thus testing whether a certain property can be violated. Symbolic execution has been used extensively in general software testing with excellent results. Remarkably, symbolic execution tools have been running 24/7 in the testing process of many Microsoft applications since 2008, revealing for instance nearly one third of the bugs discovered during the development of Windows 7, which were missed by other program analyses and blackbox testing techniques [5].

In a normal (i.e. *concrete*) execution, the program is run on specific concrete inputs, and a single control flow path is explored. In contrast, in a symbolic execution the program takes symbolic, rather than concrete, input values, and thus the exploration can proceed along several different paths simultaneously. An interpreter follows the program, assuming symbolic values for inputs rather than obtaining actual concrete values as a normal execution of the program would. It thus arrives at expressions in terms of those symbols for expressions and variables in the program, and constraints in terms of those symbols for the possible outcomes of each conditional branch.

More formally, symbolic execution is performed by a *symbolic execution engine*, that maintains for each explored control flow path:

- a *symbolic memory store* that maps variables to expressions over either symbolic or concrete values
- a first-order boolean formula representing the *path constraints*, i.e. the assumptions due to the branches taken along that particular path.

Branching updates the path constraints, whereas assignment updates the memory store.

2.1.1 Symbolic execution example

Consider the simple C code below, and assume that our goal is to determine which inputs make the assert on line 8 of function `foo` fail.

```
1 void foo(int a, int b) {
2     int x = 3, y = 0;
3     if (a != 0) {
4         y = x+1;
5         if (b == 0)
6             x = 2*(a+b);
7     }
8     assert(x-y != 0);
9 }
```

Since each input parameter can take any of the 2^{32} possible integer values, approaches like *fuzzing* that concretely run the function on randomly generated inputs will unlikely pick the exact assert-failing values. By evaluating the code using symbols for its inputs instead of concrete values, symbolic execution overcomes this limitation and makes it possible to reason in terms of *classes of inputs*, rather than single values.

More formally, every value that cannot be determined by a static analysis of the code, such as an actual parameter of a function or the result of a system call that interacts with the environment, is represented by a symbol σ_i.
At any time, the symbolic execution engine maintains a state $(stmt,\ \sigma,\ \pi)$.

- *stmt* is the next statement to be evaluated.
- σ is the symbolic store that associates program variables with expressions over concrete or symbolic values.

- π is the set of path constraints, i.e. the assumptions on the symbols σ_i due to the branches taken in the execution up to *stmt*. $\pi = true$ at the beginning of the analysis.

A graphical representation of the symbolic execution of the function `foo` is shown in Figure 2.1. Initially (execution state A) the path constraints are *true* and input arguments `a` and `b` are associated with symbolic values.
After initializing local variables `x` and `y` at line 2, the symbolic store is updated by associating `x` and `y` with concrete values 3 and 0, respectively (execution state B).
Line 3 contains a conditional branch involving symbolic values and the execution is thus forked, generating two paths. Each path gets assigned a copy of the program state at the branch instruction as well as the path constraints. Depending on the branch taken, different assumptions are then made on symbol α_a and different statements are evaluated next (execution states C and D, respectively). In the branch where $\alpha_a \neq 0$, for instance, variable `y` is assigned with `x` $+ 1$, obtaining $y \mapsto 4$ in state E, since $x \mapsto 3$ in state C. After expanding every execution state until the `assert` at line 8 is reached on all branches, we can check which input values for parameters `a` and `b` can make the `assert` fail. By analyzing execution states $\{D, G, H\}$, we can see that only H can make `x-y = 0` true. The path constraints for H at this point implicitly define the set of inputs that are unsafe for function `foo`. In particular, any input values such that

$$2(\alpha_a + \alpha_b) - 4 = 0 \wedge \alpha_a \neq 0 \wedge \alpha_b = 0$$

will make the `assert` fail. An instance of unsafe input parameters can be eventually determined by invoking a constraint solver on the path constraints, which in this example would yield $a = 2$ and $b = 0$.

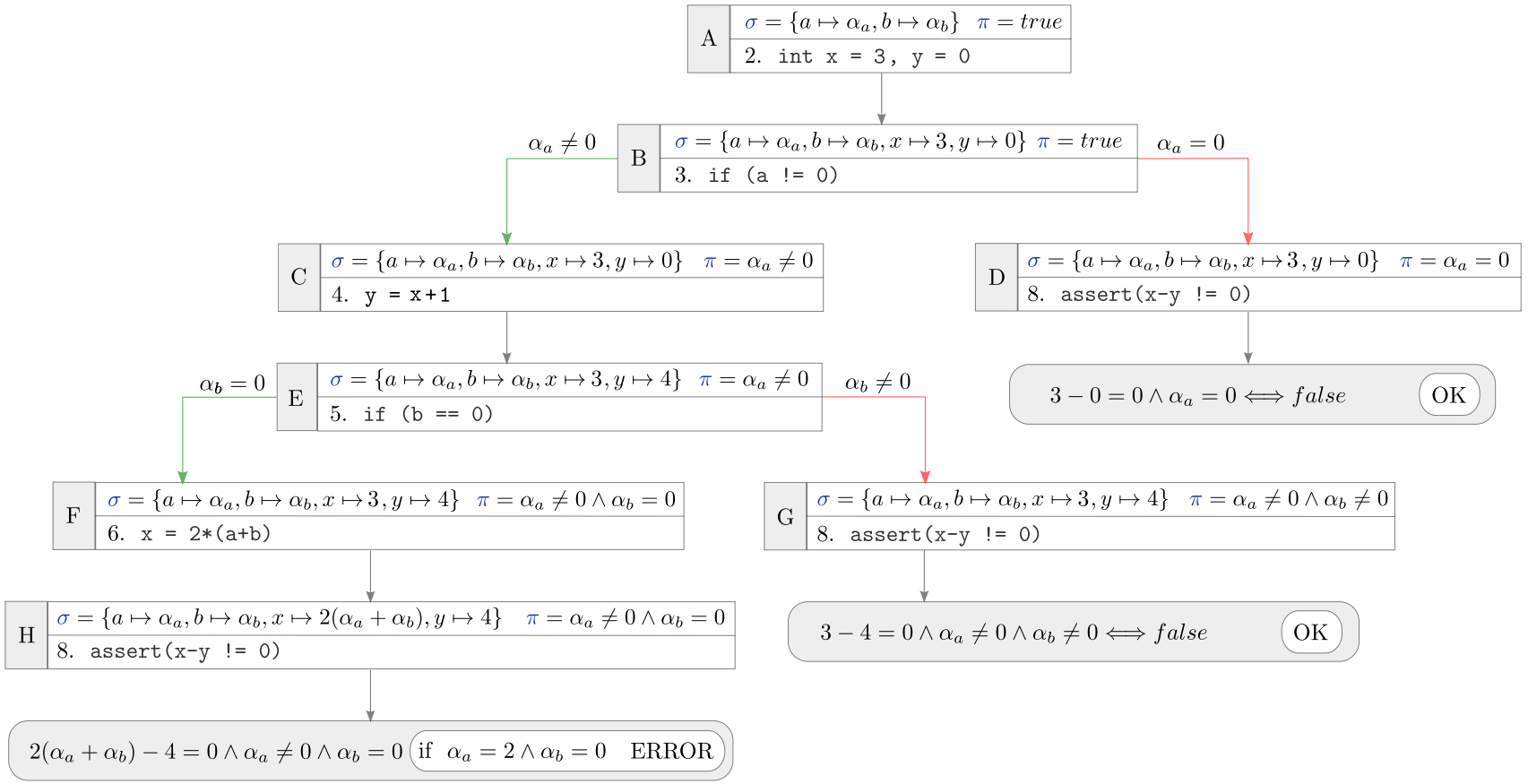


Figure 2.1. Symbolic execution tree of function `foo`

2.2 Challenges and limitations of symbolic execution

As explained in the previous sections, symbolic execution can theoretically identify all possible paths through a program. This is achieved through an exhaustive exploration of all the possible execution states and, in theory, provides a sound[1] and complete[2] methodology for any decidable analysis. Unfortunately, exhaustive symbolic execution is unlikely to scale beyond small applications. The principal challenges and limitations are listed below.

- **Path explosion**

  Symbolically executing all feasible program paths does not scale to large programs. The number of feasible paths in a program grows exponentially with an increase in program size and can even be infinite in the case of programs with unbounded loop iterations. Solutions to the *path explosion* problem generally use heuristics for path-finding to increase code coverage, reduce execution time by parallelizing independent paths, and merge similar paths.

  A special note deserve loops and, more generally, any form of iterative branching. Choosing the number of iterations to analyze is especially critical when this number cannot be determined in advance (e.g. it depends on input parameters). Unrolling iterations for every valid bound would result in a prohibitively large number of states, theoretically even infinite. Typical solutions compute an *under-approximation* of the analysis by limiting the number of iterations to some hard value k. Other more involved approaches, may try to infer loop invariants through static analysis and use them to merge equivalent states.

- **Environment Interactions**

  Programs interact with their environment (e.g. the OS, the file system, or the network) by performing system calls, receiving and handling signals, etc. Consistency problems may arise when execution reaches components that are not under the control of the symbolic execution tool (e.g., the kernel or libraries). Consider the following example:

```
1  int bar() {
2      FILE *fp = fopen("doc.txt");
```

[1] sound ≈ if it says the *property* is "true", then it is true → prevents false positives
[2] complete ≈ if the *property* is true, then it says it is "true" → prevents false negatives

```
3       ...
4       if (condition) {
5           fputs("aaa", fp);
6       } else {
7           fputs("bbb", fp);
8       }
9       ...
10      data = fgets(..., fp);
11      ...
12  }
```

This program opens a file and, based on some condition, writes different kind of data to the file. It then later reads back the written data. In theory, symbolic execution would fork two paths at line 4 and each path from there on would have its own copy of the file. The `fgets` at line 10 would therefore return data that is consistent with the value of the condition at line 4. In practice though, file operations are implemented as system calls in the kernel and are therefore outside the control of the symbolic execution tool.

The main approaches for addressing this challenge are explained below.

- **Executing calls to the environment directly**: the advantage of this approach is that it is simple to implement, we simply delegate execution to the environment. The disadvantage is that we lose state isolation: the side effects of such calls would clobber all states managed by the symbolic execution engine. In the example above, the `fgets` at line 10 would return "aaabbb" or "bbbaaa" depending on the sequential ordering of the states. This approach is therefore very rarely used.
- **Modeling the environment**: in this case, the engine instruments the system calls with a model that simulates their effects and that keeps all the side effects in per-state storage. The advantage is that we retain state isolation and get correct results when symbolically executing programs that interact with the environment. The disadvantage, though, is that we need to implement and maintain many potentially complex models of system calls. Tools such as Angr, Klee, Cloud9, and Otter take this approach by implementing models for file system operations, sockets, IPC, etc.
- **Forking the entire system state**: this is an approach employed by

symbolic execution tools based on virtual machines. They solve the environment problem by forking the entire VM state. For example, in S2E each state is an independent VM snapshot that can be executed separately. This approach alleviates the need for writing and maintaining complex models and allows virtually any program binary to be executed symbolically. However, it predictably has higher memory usage overheads (VM snapshots are large).

- **Constraint solving**
  Constraint solvers suffer from a number of limitations. They can usually handle complex constraints in a reasonable amount of time only if they are made of linear expressions over their constituents. Symbolic execution engines typically implement a number of optimizations to make queries as much *solver-friendly* as possible, for instance by performing algebraic simplifications or by splitting queries into independent components to be processed separately.
- **Binary code**
  Having the source code of an application can make symbolic execution significantly easier, as it allows the engine to exploit high-level properties (e.g. object shapes) that can be inferred statically analyzing the source code. In many cases, however, binary code is the only available representation of a program. Complexity and a lack of high-level semantics, plus the ever increasing number of complex instructions sets of modern architectures, make analyzing binary code a challenging task. A very common approach is to lift the binary code to an intermediate representation (e.g. VEX), that at the very least allows program analysis to be conducted in an architecture-independent fashion.

Summarizing, given the above limitations, in practice we often settle for slightly less ambitious goals, for example trading soundness or completeness for performance. For a detailed survey see [9].

2.2.1 A note on concolic execution

As we've discussed, symbolic execution of a program can generate, in theory, all possible control flow paths that the program could take during its concrete executions on specific inputs. While modeling all possible runs allows for very interesting analyses, it is typically not feasible in practice, especially on real-world

software, mainly because of path explosion and constraint complexity.

A standard approach to limit the resources (running time and space usage) required by the execution engine and to handle complex constraints is to mix concrete and symbolic execution: this is dubbed *concolic execution* (a portmanteau of concrete and symbolic). The basic idea is to have the concrete execution drive the symbolic execution. Besides the symbolic store and the path constraints, a concolic execution engine also maintains a concrete store. After choosing an arbitrary input to begin with, it executes the program both concretely and symbolically by simultaneously updating the two stores and the path constraints. In order to explore different paths, the path conditions given by one or more branches can be negated and the solver invoked to find a satisfying assignment for the new constraints, i.e. generate a new input, all in order to maximize code coverage.

Concolic execution, it's important to note, trades completeness for performance: false negatives are indeed possible since some paths (and therefore possibly interesting behaviors) may be missed.

2.3 Angr

In this section we will briefly present Angr, the symbolic execution engine we used to implement and test our ideas.

2.3.1 Overview

Angr [10, 12] is an open source framework for analyzing binaries developed at UC Santa Barbara. It combines both static and dynamic symbolic (concolic) analysis, shipping as a collection of Python libraries for binary analysis and dissection that make it applicable to a variety of tasks. It has been employed in a number of research works, as well as in the DARPA Cyber Grand Challenge, a two-year competition to create automatic systems for vulnerability detection, exploitation, and patching in near real-time. The simplicity of its Python APIs and the active support provided by the community make it an ideal playground for prototyping research ideas in a powerful yet simple framework.

Angr is currently under very active development [13]. For our thesis we used the newly released version 7, and that's what our description and terminology is going to refer to (frequent refactoring makes this a necessary warning). Angr itself is made up of several modules/sub-projects, all of which can be used separately

in other projects:

- *CLE*, an executable and library loader
- *archinfo*, a library describing various architectures
- *PyVEX*, a Python wrapper around the binary code lifter VEX
- *Claripy*, a constraint-solving wrapper acting as a data backend to abstract away differences between static and symbolic domains
- *angr*, the program analysis suite itself.

To give an overall idea of the inner workings: the *angr* module acts as the main interface, performing analysis and coordination activities; it loads a binary with the help of *CLE* and *archinfo*, while machine code is lifted (i.e. translated) to VEX intermediate representation through *PyVEX* as necessary. The actual analysis, finally, can be performed using the several static and dynamic symbolic analyses already available, or defining custom made ones as needed.

2.3.2 A note on *SimProcedures* and *SimStatePlugins*

A particular note deserve *SimProcedures* and *SimStatePlugins*, since they are the main components we used in the implementation of our ideas.
The `SimProcedure` class is Angr's way to define a hook to be executed during dynamic symbolic analysis. At the highest level a SimProcedure simply describes a set of actions to take on a program state (i.e. on a `SimState` object). The original and still most frequent use case for SimProcedures is to replace library functions, i.e. hooking the corresponding symbol address with an appropriate model to be invoked in its place. A *model* is a summary for a function that simulates its effects by propagating symbolic data in the same way that the original function would have [1], requiring a significantly shorter amount of time than in a symbolic exploration. Models are commonly employed in symbolic executors when dealing with the environment (as we explained in section 2.2, e.g., to simulate file system and network I/O) and also to speed up the analysis of classes of functions, such as those for string manipulation. Angr's lack of Win32 APIs models was a discouraging trait we had to face, especially in the the first stages of our experimentation with it. However, the newly released version 7 comes with an improved SimProcedure reuse (especially useful for the *libc* functions) and some initial support for the Windows APIs.

The `SimStatePlugin` class instead is, as the name suggests, a way to define a *state plugin*, i.e. a per-state storage for additional information or data. SimPro-

cedures and SimStatePlugins, in other words, respectively represent *custom action and memory elements*. Together, they are all that is necessary to model any kind of interaction with the environment, and are extensively used by angr itself to define SimLibraries that model commonly linked libraries, e.g. the C standard library *libc*.

Chapter 3

Symbolic execution of malicious software

In this chapter we will first explain exactly why evasion represents such a severe hindrance to the applicability of symbolic execution to the malware analysis domain. Then, we will give a detailed account of the most common detection techniques, presenting for each check the patch we developed in our Angr extension to bypass its exploration.

3.1 The problem with malware and evasion

Since DARPA announced in 2013 the Cyber Grand Challenge, a two-year competition to develop automatic defense systems that can discover, prove, and correct software flaws in real-time, there's been a growing interest among security professionals in the applicability of symbolic execution.
In the context of malware analysis in particular, a few earlier works have attempted to automatically explore multiple execution paths for a malware sample. In a 2007 study [2] Moser et al. present a system that can identify malicious actions carried out only when certain conditions along the execution paths are met. Brumley et al. [1, 17] have designed similar systems aiming at identifying trigger-based behavior in malware (e.g. identifying available commands for a C&C interaction). These approaches employ mixed concrete and symbolic execution to explore multiple paths, starting the execution from the very beginning of the program. Whole-code automatic analysis solutions like these, however, generally suffer from known challenges that hinder the analysis of complex malware, such as the inherent cost of constraint solving, or difficulties in handling self-modifying

code and obfuscated control flow, in addition to all the mentioned limitations associated with symbolic execution in general.
In a more recent study [11], Baldoni et al. propose a more manual approach to exploit symbolic execution for assisting malware analysis. They present an ANGR-based framework to dissect a portion of a sample that is of interest for an analyst. The solution allows the analyst to provide the symbolic engine with insights to refine and guide the analysis, provided he has sufficient knowledge to set up a minimal execution context for the exploration to start.

No matter the approach however, as explained in section 2.2, a fundamental limitation of symbolic execution is scalability. The usual heuristics and techniques designed for software testing purposes generally do not fit malware very well. In fact, while in software testing it is profitable to focus on code coverage, exploring paths capturing behaviors unexpected for a normal usage session, in the malware domain an analyst is rather interested in behaviors commonly exposed by a sample, provided that the right triggering inputs (e.g., valid commands from a C&C server) are received and/or the required environmental conditions (e.g., a specific version of Windows) are met.

In particular, the path explosion problem is especially aggravated in the malware domain by the presence of reactive-defensive evasion. Evasive checks in a malware represent critical branching points: one branch contains the malicious behavior relevant for the analysis, while the other contains generally irrelevant dissimulation code with the sole purpose of confusing and evading analysis. Clearly the exploration of the second branch represents a complete waste of resources. Though this may not seem such a big deal in and of itself, we need to keep in mind that the number of paths grows exponentially with each branch: if we have n paths reaching a branching point, passing the branch we will have $2 \times n$ paths. In case of an evasive check, of these $2n$ paths, *at most* n may actually lead to the relevant malware behavior, while instead *at least* n definitely do not contain any relevant information. Additionally, evasive checks are rarely few in number - they are usually executed in series of dozens or more, sometimes peppered through the entire executable - and most of them have an intrinsic iterative nature (e.g., performing string comparison) that aggravates the situation immensely.

3.2 Our solution

It should be now clear the threat that evasion poses to the applicability of symbolic execution in the malware domain, and thus the importance of properly handling it during the exploration. The most effective way to do just that, ideally, is to make it so that no evasive path is ever generated in the first place. That means patching the symbolic exploration so that branch points corresponding to evasive checks are not explored exhaustively as usual, but instead control flows deterministically through the *check-passed* branch. We devised and built an Angr-extension to do just that.

We researched and studied the most common reactive-detection evasive techniques employed by malware, modeled their general behavioral patterns in terms of Windows environment interactions, and built adequate patches for Angr symbolic execution engine so that it successfully passes the checks instead of exploring all branches.

To give a brief overview, the extension essentially consists of a library of patches - mainly SimProcedures that model numerous Win32 APIs - and an aptly-named *paranoid* state plugin to keep track of all evasion related information. Through the joint use of the SimProcedures and the state plugin, we hooked the corresponding Windows APIs to give a general (symbolic) behavior for most situations, but an informed anti-evasion response in case we are dealing with an evasive check.

The anti-evasion *guidance* sometimes meant adding particular constraints on otherwise purely symbolic values (e.g., memory locations or returned values), while other times meant explicitly concretizing those values. The choice was made on a per-API basis, with the overarching principle of *tweaking the symbolic execution just enough that the evasive check passes, but no more.*
It is in fact extremely important not to over-aggressively concretize the modeled APIs, especially if they are rather general ones (e.g., a `CreateFile`), since this would clearly negate the benefits of performing a symbolic execution in the first place.
Indeed, depending on the particular application of the symbolic exploration, it might pay off to modulate differently the aggressiveness of the guidance, according to the relative desirability of false negatives and false positives. Fortunately, we structured our project to be highly modifiable and extensible, hopefully making any necessary alteration as simple to implement as possible.

3.2.1 A note on *StdcallSimProcedures* and *ParanoidPlugins*

In this subsection we will spend a couple of words to give a brief description of two simple yet fundamental custom components we defined, that are employed very frequently in our extension: `StdcallSimProcedure` and `ParanoidPlugin`.

`StdcallSimProcedure` is a dummy subclass of angr's `SimProcedure`. It does not introduce any additional property. Instead its sole purpose is to act as a tag to identify SimProcedures that should be invoked using the *stdcall* calling convention. Angr defines *cdecl* (caller-cleanup) as the standard calling convention for x86 architectures. The majority of Windows APIs, however, use *stdcall* (caller-cleanup) as the standard calling convention. A mismatch in calling conventions is no laughing matter, as it produces invalid stack configurations resulting in silent errors and random bugs. We introduced the `StdcallSimProcedure` to explicitly tag those models that require a *stdcall* convention, additionally making it possible to automate the hooking of the corresponding symbols (performed by an auxiliary `hook_all` function).

`ParanoidPlugin` instead is a normal subclass of angr's `SimStatePlugin`. It is registered with (and accessible by) the name *paranoid* in each program state (i.e. `SimState` object) and we use it as additional per-state storage in which to keep track of all evasion related information.

3.3 Evasive checks models and patches

In this section we will present the most commonly employed reactive-detection evasive techniques that we studied, their general behavioral patterns (especially in terms of Windows environment interactions), and the way we patched them in our Angr extension so that dynamic symbolic execution successfully passes the checks instead of exploring each branch. The structure of the narration will follow the reactive-defensive evasion categorization introduced in section 1.2. For each category we will give an introductory, more detailed analysis of an archetypal instance, and then follow up with a more concise account for the other techniques.

3.3.1 OS artifacts detection

In this section we will talk about evasive techniques targeting artifacts produced in the operating system of the machine hosting the analysis product.

Windows Registry artifacts

Registry artifacts are definitely the most common type of OS artifacts checked by malicious software. The *Windows Registry* contains a wealth of information, settings, options, and other values for programs and hardware installed on the system. Registry keys then represent an authentic gold mine for evasive checks.

There are two kind of patterns for registry checking:

- Registry key *presence* checking, used when the presence of the key itself reveals the presence of the corresponding application on the system (e.g., `"HKEY_LOCAL_MACHINE\SOFTWARE\VMware, Inc.\VMware Tools"` revealing we are running in a VMware [30] virtual machine).
- Registry key *value* checking, used for general keys whose particular values can reveal the presence of an application (e.g., `"HKEY_LOCAL_MACHINE\HARDWARE\Description\System\SystemBiosVersion"`'s value being `"BOCHS"`, revealing that we are running on the Bochs [16] IA-32 emulator).

Registry key presence checking Windows `RegOpenKeyEx` API is used to obtain an handle to the registry key, and thus implicitly verify its presence. Consider for example the following self-explanatory C code fragment:

```
1  int vbox_check() {
2      HKEY regkey;
3      LONG ret;
4      char regkey_s[] = "SOFTWARE\\Oracle\\VirtualBox Guest Additions"
5      ret = RegOpenKeyEx(hKey, regkey_s, 0, KEY_READ, &regkey);
6      if (ret == ERROR_SUCCESS) {
7          RegCloseKey(regkey);
8          return TRUE;
9      }
10     else
11         return FALSE;
12 }
```

To bypass this kind of check, in our extension we define a list of blacklisted analysis-related strings (e.g., `"vbox"`, `"vmware"`, `"sandboxie"`, etc). We then hook `RegOpenKeyEx` with a SimProcedure that returns a concrete `ERROR_FILE_NOT_FOUND` value if the requested key contains a blacklisted string, otherwise a symbolic handle.

The abridged Python code for the *SimProcedures* is reported below.

```
1  class RegOpenKeyExA(StdcallSimProcedure):
2      def extract_string(self, addr):
3          return self.state.mem[addr].string.concrete
4
5      def run(self, hKey, lpSubKey, ulOptions, samDesired, phkResult):
6          ...
7          regkey_name = self.extract_string(lpSubKey).upper()
8          analysis_related = any(
9            ana_str in regkey_name.lower() for ana_str in ANALYSIS_STRS)
10         if analysis_related:
11             ret_expr = 2  # ERROR_FILE_NOT_FOUND
12         else:
13             handle = self.state.solver.BVS(
14               "handle_{}_{}".format(self.display_name, regkey_name), 32)
15             ...
16             self.state.memory.store(
17               phkResult, handle, endness=self.arch.memory_endness)
18             ...
19             self.state.paranoid.open_regkeys[handle] = regkey_name
20             ...
21             ret_expr = self.state.solver.BVS(
22               "retval_{}_{}".format(self.display_name, regkey_name), 32)
23             ...
24         return ret_expr
25
26 class RegOpenKeyExW(RegOpenKeyExA):
27     def extract_string(self, addr):
28         return self.state.mem[addr].wstring.concrete
```

A couple of additional observations: for APIs involving strings (either as input or as output), Windows generally provides two separate symbols, one for ANSI 8-bit character strings (ending in A) and one for Unicode 16-bit *wide* character strings (ending in W)[1]. As shown in the Python snippet above, we can elegantly handle this dual-versioning through subclassing and the overloading of auxiliary string extractor or constructor methods.

Secondly, notice on line 19 how we keep track of the mapping between the returned symbolic regkey handle and the name of the corresponding registry key by storing it in our *paranoid* plugin. Aside from debugging purposes, this proves useful for

[1]Unicode, in particular UTF-16, is Windows preferred character encoding

intelligently modeling other registry related APIs, that rely on previously opened handles (e.g., `RegQueryValueEx` described below).

Registry key value checking For registry key *value* checking, the check pattern is slightly more complex: Windows `RegOpenKeyEx` API is used to obtain an handle to the registry key; this handle is then passed to `RegQueryValueEx` along with the field name we are interested in querying. The returned field value is then compared with some relevant defaults in order to detect the corresponding analysis product.

To bypass this pattern, in our extension we define a dictionary of *sensitive* keys and corresponding concrete *safe* values. We then hook `RegQueryValueEx` to return the defined concrete value for sensitive keys (exploiting the handle → regkey name mapping in our *paranoid* plugin), otherwise a purely symbolic value.

User name check

Analysis environment often use standard names for the user account, like *sandbox*, *malware*, or *virus*. The `GetUserName` API is invoked to obtain the user name associated to the current process, and then this is compared to common analysis-related names.

To bypass such a simple check all we had to do in our extension is hook `GetUserName` with a SimProcedure that returns a safe, concrete name, e.g. `"John"`.

Process enumeration check

A malware can potentially decide to enumerate all processes running on a machine, searching for analysis-related ones. To do this it invokes Windows `CreateToolhelp32Snapshot` function with the `TH32CS_SNAPPROCESS` flag to obtain an handle to a system snapshot of all running processes. It then scans through the processes using the `Process32First` and `Process32Next` APIs, looking at the process executable file names.

This kind of check, though not very common, is very troublesome for a symbolic engine because of the unbounded iterative nature of the process enumeration pattern. It is fundamental then to streamline the exploration of this check as much as possible. We could define a concrete list of running processes and set up the aforementioned APIs to use this list. However, since those APIs are not commonly used for purposes other than process enumeration, in our extension we adopted a lazier but practically equally efficient approach: we

simply hooked `CreateToolhelp32Snapshot` with a SimProcedure always failing the call (i.e. returning a concrete `INVALID_HANDLE_VALUE`) if it's invoked with the `TH32CS_SNAPPROCESS` flag. The check then would not actually be able to enumerate the processes and thus succeed automatically. Clearly though, as with any other proposed solution, exceptions may apply, and changes may thus be necessary.

Windows check

Windows `FindWindow` API retrieves an handle to a window whose name or class matches a specified string. Malwares are known to abuse this API to perform evasive checks, calling it repeatedly to verify the presence of windows corresponding to analysis-related programs. For example, searching for a window whose name is `"VBoxTrayToolWnd"` allows the malware to detect if it is running on a VirtualBox [24] machine.

We bypass this check hooking `FindWindow` with a SimProcedure that scans the specified string for the presence of blacklisted analysis-related keywords. If such keywords are detected, then it returns a concrete `NULL` value, indicating that no windows matching that name were found, otherwise it returns a symbolic handle.

3.3.2 Process artifacts detection

In this section we will talk about evasive techniques targeting artifacts related to the process that is hosting the executable.

Hooks check

Many analysis environments hook Windows APIs in order to track a program interaction with the system. Many evasive malwares then check for this hooking in order to escape detection. There are several ways APIs can be hooked, e.g. *IAT hooking* (overwriting the address of the API in the Import Address Table with the custom function's address), or *inline hooking* (overwriting the actual API function code contained in the imported DLLs, for example injecting an unconditional jump to the analysis code in the first instructions). No matter the particular technique that is used to implement the hook, the net result is always the same: the custom code differs from the original genuine API code, and that's exactly what evasive techniques check.

A common pattern is to read the first bytes of the API code in memory and compare them with known genuine values. Consider the following C code proof of concept:

```
1 int is_CreateProcessA_hooked() {
2     BYTE *b = (BYTE *)CreateProcessA;
3     if ((*b == 0x8b) && (*(b+1) == 0xff))
4         return FALSE;
5     else
6         return TRUE;
7 }
```

The check compares the first two bytes of the `CreateProcessA` API with the known genuine value `0x8bff`. `0x8bff` is the opcode for a `MOV EDI, EDI`, essentially a two-byte `NOP` present in all Windows APIs to allow for the *hot-patching* of the function (a form of inline hooking intentionally supported by the Windows API). If `0x8bff` is not present, then a hook is definitely in place and the check detects it.

Note that Angr, working as a kind of interpreter/emulator, doesn't need to inject code in order to hook anything; it just hooks the address, i.e. it controls access to the address and executes the hooking function. This kind of check may still represent a problem though. In fact, if we don't load the required DLLs along with the binary, all unresolved symbols are provided by an `ExternObject`, a binary object that the CLE loader uses to map unresolved symbols (thus allowing them to be hooked). Being a simple aid, the `ExternObject` is zero filled. Therefore an hook check on an unresolved symbol would just retrieve a bunch of zeros, causing the check to deterministically fail.

To solve this problem we could patch the `ExternObject`, either filling it with symbolic values (so that symbolic execution explores both paths of the hook check), or writing valid opcodes at the addresses interested by the check (so that we bypass the check fail path).

A better, more robust solution though, and the one we employ in our extension, is to simply load all the required DLLs along with the binary. This way the CLE loader creates a binary object filled with the right code for each DLL, and thus any hook check would automatically pass.

Read section A.3 for additional details and a guide.

Injected DLLs check

Similarly to API hooking, many analysis products instrument the process of a potentially malicious sample injecting custom DLLs in its address space so as to track, alter, or extend its functionalities. A malware can then check for the presence of extraneous DLLs in its address space in order to detect deliberate instrumentations (e.g., the presence of `sbiedll.dll` reveals that the process has been instrumented by the Sandboxie [27] security program).

Windows `GetModuleHandle` API is used to check for the presence of a previously loaded module. To bypass this check then we could define a list of blacklisted modules, and hook `GetModuleHandle` with a SimProcedure that returns a concrete `NULL` value if the requested module is a blacklisted one (meaning it was not found), otherwise a symbolic handle.

Alternatively, in our extension, since we are loading all the program required DLLs along with the main binary , we just perform a concrete check, i.e. hook `GetModuleHandle` with a SimProcedure that concretely verifies the presence of the specified module among the loaded objects. Since we load all and only the modules required by the program itself, no extraneous module could ever be present and any check would always pass automatically.

Debugging checks

Evasive samples can exploit many available Windows APIs in order to check for the presence of a debugging tool attached to the running process.

- `IsDebuggerPresent`
  returns a boolean value indicating whether the calling process is running under a user-mode debugger.

  In our extension we simply hook it with a SimProcedure always returning false.

- `OutputDebugString`
  sends a string to the debugger for display. Checking the last error value then it is possible to determine the presence of a debugger: the last error value is cleared before the call (invoking the `SetLastError` API), and is checked for change after the call (invoking the `GetLastError` API). If no error occurred, then the `OutputDebugString` invocation was successful, implicitly revealing that a debugging tool is indeed present.

  To patch this behavior, we store the *Last-Error* value in our *paranoid* plugin.

Then we hook `SetLastError` and `GetLastError` with SimProcedures that just set and get the value in the plugin, and `OutputDebugString` with a SimProcedure that fails the call and modifies the *Last-Error* value.

- `CheckRemoteDebuggerPresent`
  populates a boolean value indicating whether the specified process is being remotely debugged (i.e. is being debugged by a debugger residing in a separate and parallel process).

  In our extension we simply hook it with a SimProcedure always returning false.

3.3.3 File system artifacts detection

In this section we will talk about evasive techniques targeting artifacts present in the file system of the hosting machine.

File system artifact presence check

Practically all applications store data in the file system, and security-related products are no exception. An evasive malware can then look at the files present on the machine in order to determine if it is in an hostile environment.

Some commonly checked files are

- executables, e.g. `C:\WINDOWS\System32\VBoxService.exe` (revealing that we are running in a VirtualBox VM)
- drivers, e.g. `C:\WINDOWS\System32\drivers\vmhgfs.sys` (revealing that we are running in a VMware VM)
- directories, e.g. `C:\Program Files\Oracle\VirtualBox Guest Additions\` (revealing we are in a VirtualBox VM)
- DLLs, e.g. `C:\WINDOWS\System32\snxhk.dll` (revealing the presence of Avast [14] antivirus products)
- virtual devices, e.g. `\\.\pipe\VBoxTrayIPC` (revealing we are in a VirtualBox VM)

The two main patterns for file system checking use the following APIs:

- `GetFileAttributes`
  to retrieve the file system attributes for a specified file. Implicitly then, if the call succeeds, the file exists and thus the related program is installed on the machine. Consider the following C code example:

```
int vmware_check() {
    char file[] = "C:\\WINDOWS\\system32\\drivers\\vmmouse.sys"
    DWORD res = INVALID_FILE_ATTRIBUTES;
    res = GetFileAttributes(file);
    if (res != INVALID_FILE_ATTRIBUTES)
        return TRUE;
    else
        return FALSE;
}
```

To bypass these checks, in our extension we simply hook `GetFileAttributes` with a SimProcedure returning a concrete `INVALID_FILE_ATTRIBUTES` value if the requested file is a blacklisted one (i.e. analysis-related strings present in the file path), a symbolic value otherwise. The abridged Python code for the *SimProcedure* is reported below.

```
class GetFileAttributesA(StdcallSimProcedure):
    def extract_string(self, addr):
        return self.state.mem[addr].string.concrete

    def run(self, lpFileName):
        ...
        malware_related = any(
          mal_s in file_name.lower() for mal_s in MALWARE_STRS)
        analysis_related = any(
          vm_s in file_name.lower() for vm_s in ANALYSIS_STRS)
        if malware_related or analysis_related:
            ret_expr = -1  # INVALID_FILE_ATTRIBUTES
            # set last error value to ERROR_FILE_NOT_FOUND
            self.state.paranoid.last_error = 0x2
        else:
            ret_expr = self.state.solver.BVS(
              "retval_{}_{}".format(self.display_name, file_name), 32)
        ...
        return ret_expr

class GetFileAttributesW(GetFileAttributesA):
    def extract_string(self, addr):
        return self.state.mem[addr].wstring.concrete
```

- `CreateFile` with read access flag
  to open the specified file for reading. If the call succeeds, then the file exists and the related program is likely installed on the machine.

  To patch this kind of pattern, we hook the `CreateFile` API with a SimProcedure that fails, returning an `INVALID_HANDLE_VALUE` and setting *LastError* to `ERROR_FILE_NOT_FOUND`, if the requested file is a blacklisted one, otherwise it returns a symbolic handle.

Execution path check

As for user names, analysis environment often use standard execution path names, for example containing strings like `"sandbox"`, `"malware"`, `"virus"`, or `"sample"`. The `GetModuleFileName` API is invoked to retrieve the fully qualified path of the executable file, and this is checked for the presence of analysis-related strings.

To patch this kind of check, in our extension we hook `GetModuleFileName` with a simple SimProcedure that returns a safe concrete path, i.e. not containing any analysis-related strings, e.g. `"C:\installer.exe"`.

Common file names check

Similar to the execution path check. The available drives are checked for the presence of files with an analysis-related name, e.g. `"sample.exe"` or `"malware.exe"`. The usual pattern consists in obtaining a list of the logical drives root directories through the `GetLogicalDriveStrings` API, and checking these for the presence of suspicious files invoking the `GetFileAttributes` API.

In our extension we patch this kind of check adding a simple hook for `GetLogicalDriveStrings` that concretizes the drive string list, returning just a single directory, `"C:\"`. The SimProcedure hooking the `GetFileAttributes` we described in a previous section does the rest.

3.3.4 Hardware-based detection

In this section we are going to look at the most common hardware-based detection techniques, i.e. techniques that exploit hardware differences between an analysis environment and a normal one.

Overly modest hardware checks

Analysis environments, in order to optimize resource sharing, generally allocate as few resources as possible to each sandbox, creating hardware configurations that are too modest when compared to current standards. These peculiarities are exploited by malicious software to perform evasive checks.

Single-CPU check Windows `GetSystemInfo` API populates a `SYSTEM_INFO` struct with information on the current computer system architecture. In particular, it specifies the number of processors available on the machine. Consider then the following simple evasive check:

```
int single_cpu_check() {
    SYSTEM_INFO sysInfoStruct;
    GetSystemInfo(&sysInfoStruct);
    if (sysInfoStruct.dwNumberOfProcessors < 2)
        return TRUE;
    else
        return FALSE;
}
```

The underlying semantic of the check is sensible: modern machines are practically always multi-core, whereas analysis environments typically allocate just one core per sandbox.

To patch this check we may be tempted to hook `GetSystemInfo` with a SimProcedure returning a completely concrete instance of the `SYSTEM_INFO` struct, i.e. giving actual concrete values to each field. Indeed this would be a viable solution, at least to pass this particular check. In our extension however, we follow the general policy of preserving the indeterminacy and refrain from overly-aggressive concretization whenever possible. To pass this check then, and generally any check involving a structured value, we constrain only the relevant fields, leaving all others as purely symbolic. In this particular case, we concretize only the `dwNumberOfProcessors` field with a safe value of 4. The abridged Python code for the SimProcedure is shown below.

```
class GetSystemInfo(StdcallSimProcedure):
    def run(self, lpSystemInfo):
        ...
```

```
        sysinfo_struct = self.state.solver.BVS('SYSTEM_INFO', 36 * 8)
        self.state.memory.store(lpSystemInfo, sysinfo_struct)
        dwNumberOfProcessors = self.state.solver.BVV(4, 4 * 8)
        self.state.memory.store(lpSystemInfo+20, dwNumberOfProcessors)
        ...
        return
```

Small amount of RAM check Automated analysis environment generally allocate no more than one or two GBs of RAM to each virtual machine. On the other hand, modern machines nowadays typically have at the very least 4 GBs of volatile memory. A good discriminant for an evasive check.

The `GlobalMemoryStatusEx` API is invoked to populate a `MEMORYSTATUSEX` struct with information about the current state of both physical and virtual memory, in particular the amount of total physical memory in bytes. This value is then compared with an empirical threshold, e.g. 1 GB.

To patch it, in our extension we just hook `GlobalMemoryStatusEx` with a SimProcedure populating the `ullTotalPhys` field of the `MEMORYSTATUSEX` struct with a concrete, sufficiently large value, e.g. 8 GBs.

Small drive size check Analysis environments, again to optimize resource sharing, generally don't allocate much space for the permanent storage, typically no more than a dozen GBs. On the other hand, storage space is ever cheaper nowadays, so real users presumably have considerably sized disks. Another perfect discriminant for evasive checks.

Several ways exist to obtain the available drive space. The two main ones use the following APIs:

- `DeviceIoControl` with control code `IOCTL_DISK_GET_LENGTH_INFO`
- `GetDiskFreeSpaceEx`

Both APIs populate a `GET_LENGTH_INFORMATION` struct with the disk size in bytes.

To patch this kind of checks then, we hook both `DeviceIoControl` and `GetDiskFreeSpaceEx`, populating the `GET_LENGTH_INFORMATION` struct with a concrete, large enough value, e.g. 256 GBs.

Hardware fingerprinting checks

Emulation and virtualization software often fingerprints hardware devices information, unwittingly producing attractive targets for evasive checks.

CPUID-based checks The `CPUID` assembly instruction allows software to discover details of the machine processor. Since most virtualization environments fingerprint the info (e.g., the vendor name), it is a clear target for evasive checks. For example, in a QEMU [26] virtual machine `CPUID` returns `"QEMU Virtual CPU"` as the CPU brand.

Angr implements the `CPUID` instruction as a *VEX dirty helper*, i.e. an helper function internally invoked by the dynamic exploration engine when the `CPUID` instruction is encountered in the VEX IR. Fortunately, the dirty helper concretizes all info emulating a real CPU; therefore no additional instrumentation is required to handle these checks.

Network adapter check A very targeted kind of check compares the network adapter details with some known default values typical of specific virtualization softwares. For example VirtualBox machines MAC addresses are known to start with `"\x08\x00\x27"`.
A similar check scans the network adapter description for the presence of known strings, like `"VMware"` or `"VBox"`.

Network adapters information is obtained invoking the `GetAdaptersAddresses` API, that populates a `IP_ADAPTER_ADDRESSES` struct (a linked list) with the adapters details. The iterative nature of the check (traversing the linked list structure) makes it a significant danger for purely symbolic exploration.

To patch this kind of check, we populate the `IP_ADAPTER_ADDRESSES` struct with symbolic values, except for those fields relevant to the evasive checks, which we concretize with innocent common values. In particular, we concretize the `Description` field with a default `"Intel(R) Gigabit Network Connection"`, and the MAC address field with a random 6 bytes value. A fair warning when handling linked list structures: aside from the fields relevant for the check, it is fundamental to constrain the fields used to traverse the list, in particular those indicating the end of the list. Failing to do so, given the iterative nature of the traversing, would generate an extremely dangerous unbounded loop. In the particular case of `IP_ADAPTER_ADDRESSES` struct, this meant concretizing the `Next` field of the single instance we return with a `NULL` (i.e. 0) value.

3.3.5 Time-based detection

In this section we are going to look at the most common time-based detection techniques, i.e. techniques that detect an analysis environment exploiting timing discrepancies with a normal environment. These differences are practically always present, either due to overhead or explicit modifications and patching.

Sleep patched check

Automated analysis environments, to optimize resource sharing, generally execute a suspicious program for a limited amount of time, typically no more than a couple of minutes. Many malwares then try to proactively stall, waiting for a sufficiently large interval of time before doing anything harmful, in order to appear innocuous to the analysis environment. Many analysis environments, as a result, patch the sleeping/waiting functions abused by malware, in particular Windows `Sleep` API. This in turn lends itself to a very reliable evasive check: the malware invokes the `GetTickCount` API to get the number of elapsed milliseconds since the system was started; it then sleeps for a certain amount of time; finally upon waking up it calls `GetTickCount` again and verifies whether the elapsed number of milliseconds approximately coincides with the sleep period. Consider the following proof of concept:

```
1  int is_sleep_patched() {
2      DWORD time1, time2;
3      time1 = GetTickCount();
4      Sleep(500);
5      time2 = GetTickCount();
6      if (time2 - time1 > 450)
7          return FALSE;
8      else
9          return TRUE;
10 }
```

To patch this check, we hook both `GetTickCount` and `Sleep`: we make `GetTickCount` return a concrete value stored in our *paranoid* plugin, whereas `Sleep` simply increases this value by the specified amount. The abridged Python code is shown below.

```
class Sleep(StdcallSimProcedure):
    def run(self, dwMilliseconds):
        ...
        self.state.paranoid.tsc += (
          self.state.solver.eval(dwMilliseconds) * TICKS_PER_MS)
        ...
        return

class GetTickCount(StdcallSimProcedure):
    def run(self, ):
        ...
        ret_expr = self.state.paranoid.tsc // TICKS_PER_MS
        # additionally increase the tick counter to
        # handle repeated GetTickCount calls check
        self.state.paranoid.tsc += TICKS_PER_MS
        ...
        return ret_expr
```

An additional observation: in our *paranoid* plugin we actually store a Time Stamp Counter value, representing the number of ticks - or processor cycles - since the system start. This value is converted in milliseconds according to Windows defined constant `TicksPerMillisecond = 10000`. We use this representation instead of a simpler millisecond-based one in order to have a single, coherent view of time for all time-based patches (the TSC value is necessary to handle `RDTSC`-based checks described later).

Uptime check

Automated analysis environments, as discussed, execute a program for a limited amount of time. At the end of the analysis they reset the virtual machine, and carry on analyzing other samples. This implies that sandboxes generally have a very small uptime, typically in the order of minutes. On the other hand, a real machine nowadays is hardly ever shutdown anymore, thus its uptime is very likely to be a lot larger. A simple but reliable evasive check then consists in comparing the machine uptime, obtained by invoking the aforementioned `GetTickCount` API, with an empirical threshold, e.g. 15 minutes, and discriminate accordingly.

To pass this check, we just make sure to initialize the concrete value for the TSC stored in our *paranoid* plugin with a sufficiently large value, e.g. 50 minutes.

RDTSC-based checks

The `RDTSC` assembly instruction reads the Time Stamp Counter (TSC) register, returning the number of cycles since reset in the `EDX:EAX` registers. The fine granularity of the TSC register is exploited in evasive checks: `RDTSC` is typically used to calculate the elapsed time between the start and the end of a small code block execution; since code inside a virtual machine typically runs slower than on a real one, a larger than usual differential (i.e. greater than an empirical threshold) is a strong indication of VM presence.

To pass `RDTSC`-based checks, in our extension we concretize the read TSC, storing it in the *paranoid* plugin, and increment it by a sufficiently small value at each invocation of the `RDTSC`, so that the differential is always positive but smaller than any threshold. Simple tests revealed ∼500 to be the differential between successive invocations of the `RDTSC` instruction on several machines, and 700/750 were the thresholds found in several `RDTSC`-based checks, so 500 was chosen as the increment.

An additional implementation note: the `RDTSC` instruction, like the previously described `CPUID` is implemented by Angr as a *VEX dirty helper*, i.e. a function internally invoked by the engine when the `RDTSC` instruction is encountered in the VEX representation. Since Angr does not offer an interface to modify or redefine these dirty helpers, to hook it we *monkey patch* the `angr.engines.vex.dirty` module, substituting in our custom `RDTSC` helper in place of the default one.

3.3.6 Reverse Turing tests

Reverse Turing tests, i.e techniques that check for the presence of a human user, are generally very effective ways to discern between an automated analysis environment and a normal one. However, malicious software does not often employ such techniques since they generally require user interaction and thus may very easily arouse suspicion. The few techniques that are used are typically of the *silent* type, e.g. passive user input detection, or file system *wear-and-tear* assessment. Of these, the only one we actually saw in real samples was the simple mouse movement check.

Mouse movement check

This is a very simple check: the `GetCursorPos` API is invoked to get the current coordinates of the mouse cursor; the process then sleeps for a couple of seconds;

finally the `GetCursorPos` is invoked again and the new coordinates are compared with the old ones to reveal mouse movement. The underlying idea is that on a real machine the user would be actively using the mouse and thus cursor movement would be present, whereas on an automated analysis environment the mouse would not be used at all, if even present in the first place. A simple C code proof of concept follows below.

```
1  int is_mouse_moved() {
2      POINT position1, position2;
3      GetCursorPos(&position1);
4      Sleep(5000);
5      GetCursorPos(&position2);
6      if (position1.x==position2.x && position1.y==position2.y)
7          return FALSE;
8      else
9          return TRUE;
10 }
```

To patch this kind of check, in our extension we simply hook the `GetCursorPos` API with a SimProcedure that returns random coordinates at each invocation.

3.4 Conclusions

In this chapter we have presented the most commonly employed reactive-detection evasive techniques that we studied, in particular focusing on their general behavioral patterns in terms of Windows API interactions. For each technique analyzed we described the way we patch it in the Angr extension we developed so that the symbolic execution successfully passes the check instead of exhaustively exploring each branch. A summary is reported in Table 3.1.

It is important to notice that the list of covered techniques should not be considered in any way exhaustive, nor did we have any ambition for it to be. Malware researchers admittedly track more than 500 evasive behaviors used to bypass detection [19]. The malware domain is full of tricks that exploit transient vulnerabilities in software, and evasion makes no exception. We shunned these techniques in our study since they are not robust, they are not stable, and they are generally patched as soon as they are discovered or they become sufficiently widespread.

In our study we focused on the most *consistent and widespread* detection techniques. However, other techniques do exist, and they are indeed used by malware, though less frequently. With this in mind, we designed the entire project to be easily modified and expanded, hopefully making any necessary addition or alteration as simple to implement as possible.

Table 3.1. Analyzed evasive checks and patches summary

Category Check	Patched APIs
OS artifacts detection	
Registry key presence	`RegOpenKeyEx`
Registry key value	`RegOpenKeyEx` `RegQueryValueEx`
User name	`GetUserName`
Process enumeration	`CreateToolhelp32Snapshot`
Windows	`FindWindow`
Process artifacts detection	
Hooks	-[1]
Injected DLLs	`GetModuleHandle`[1]

Debugging	`IsDebuggerPresent`
	`OutputDebugString`
	`CheckRemoteDebuggerPresent`

File system artifacts detection	
File system artifact presence	`GetFileAttributes` `CreateFile`
Execution path	`GetModuleFileName`
Common file names	`GetLogicalDriveStrings` `GetFileAttributes`

continued ...

... continued

Category Check	Patched APIs
Hardware-based detection	
Single-CPU	`GetSystemInfo`
Small amount of RAM	`GlobalMemoryStatusEx`
Small drive size	`DeviceIoControl` `GetDiskFreeSpaceEx`
CPUID fingerprinting	`CPUID`[2]
Network adapter details	`GetAdaptersAddresses`
Time-based detection	
Sleep patched	`GetTickCount` `Sleep`
Uptime	`GetTickCount`
RDTSC timing	`RDTSC`
Network adapter details	`GetAdaptersAddresses`
Reverse Turing tests	
Mouse movement	`GetCursorPos`

1 No patch necessary with DLLs loading.

2 Angr's default dirty helper is enough.

Chapter 4

Experimental Evaluation

In this chapter we will report the results obtained during the experimental evaluation of the Angr extension we developed in order to patch the symbolic exploration of evasive checks.

We tested the extension first on *Paranoid Fish*, an open source demonstration tool that employs many techniques to detect sandboxes and analysis environments in the same way malware families do. Given the availability of the source code of all the checks, pafish represented an ideal test bed for the extension, especially in the initial stage of its development.
Later, we also tested the project on real malware samples, in order to effectively assert its applicability to real-life scenarios.
All tests were conducted on a physical machine with the following specifications:

CPU	:	Intel i7-6700HQ @ 2.60 GHz
RAM	:	16 GBs
OS	:	Ubuntu 17.04
Python	:	2.7.13
Angr	:	7

4.1 Paranoid Fish

Paranoid Fish (also known as *pafish* for short) is an open source tool that demonstrates several techniques employed by malware families in order to detect whether they are being executed in an analysis environment, be it a debugger, a virtual machine or a sandbox. The project is hosted on GitHub [25]. Developed in C, it is divided in several modules corresponding to a rough categorization of the checks they encompass:

- `debuggers.c` (debugger detection)
- `cpu.c` (CPU instruction-based VM detection)
- `gensandbox.c` (generic sandbox detection techniques)
- `hooks.c` (hooking detection)
- `vbox.c` (VirtualBox [24] virtualization detection)
- `vmware.c` (VMware [30] virtualization detection)
- `qemu.c` (QEMU [26] emulator detection)
- `bochs.c` (Bochs [16] emulator detection)
- `sandboxie.c` (Sandboxie [27] detection)
- `cuckoo.c` (Cuckoo Sandbox [18] detection)
- `wine.c` (Wine [31] compatibility layer detection)

Each check is nicely encapsulated in a function returning `TRUE` (i.e. 1) if the check succeeded in detecting the analysis product, `FALSE` (i.e. 0) otherwise. Pafish invokes each check during the execution of its main module, producing a detailed report on the command line.

4.1.1 Evaluation

For initial testing purposes, we were interested in executing each check in isolation, carefully controlling whether the symbolic execution would pass the check, producing only paths returning `FALSE`. To do this we had to provide angr with the address of each check we wanted to perform. Rather than proceed manually (i.e. tediously gathering each check address from the disassembly), given the availability of the source code, we compiled pafish leaving all symbol information intact (i.e. removing the `-s` linking flag from the makefile that would otherwise strip all symbols from it).
On Linux systems the command `nm --demangle` can then be used to obtain the demangled symbol information from a binary file. We wrote a simple Python script that given a binary and a list of symbol names, produces a JSON object containing a list of (symbol_name, symbol_address) tuples. We then used the script to produce a list of all the checks in the pafish binary, imported this list in our angr test script and proceeded with checks exploration.

Each check was executed firstly without the aid of the developed extension: in particular all the required Windows DLLs were loaded, and each relevant Windows API was modeled with angr's corresponding SimProcedure if available, otherwise

with a `ReturnUnconstrained` stub, i.e. a SimProcedure that simply returns an unconstrained, purely symbolic value. The same check was then executed applying our extension. The results are reported in the table below. For each check we report the number of generated paths with and without the aid of our extension, specifying for each the portion of successful paths (i.e. those returning `FALSE`, indicating that the check passed) and the portion resulting in a detection (i.e. those returning `TRUE`).

It is important to notice that most of the reported values for unaided execution are strongly dependent on the particular optimizations introduced by the angr engine and the various execution bounds applied by the available angr's SimProcedures (e.g., most of the C standard library SimProcedures introduce hard bounds on the maximum length of a symbolic string). In fact, because of the iterative nature of many checks, a purely symbolic exploration would actually never terminate (for example iterating on all possible lengths of a symbolic string). In other words, we report the results obtained without the aid of our extension not to quantify precisely the number of paths generated by a purely symbolic execution (which would not make much sense anyway, since there is no *purely symbolic executor* - all symbolic execution engines introduce many heuristics and empirical optimizations in order to mitigate the scalability problems we described in section 2.2). Instead we report them to motivate the need for our extension, clearly but simply showing how intractable a symbolic execution of the entire section of checks would be even given the amount of optimizations built into angr already.

Finally note that the ideal scenario for execution with our extension would have just a single path generated by the exploration of the check, returning `FALSE`. This, in fact, would mean that the check exploration was effectively bypassed, and no unnecessary paths were generated by its execution. However, sometimes the number of paths generated may be more than one, though each of them still returning `FALSE`. This is usually due to some kind of *accessory checking*, e.g. checking whether an auxiliary function returned successfully. This accessory checking does not ultimately change the return value of the check, but it still corresponds to different execution paths. It is therefore generally normal to have a single check generate multiple paths returning the same value: as long as the return value is `FALSE`, we are successfully passing the check, though we may be doing so *in more than one way.*

Table 4.1. Paranoid Fish checks evaluation

Module name	unaided			extended		
Check function	Tot	T	F	Tot	T	F
debuggers.c						
debug_isdebuggerpresent	2	1	1	1	0	1
debug_outputdebugstring	2	1	1	1	0	1
cpu.c						
cpu_rdtsc	4	3	1	1	0	1
cpu_rdtsc_force_vmexit	4	3	1	1	0	1
cpu_hv	1	0	1	1	0	1
cpu_known_vm_vendors	1	0	1	1	0	1
gensandbox.c						
gensandbox_mouse_act	3	1	2	1	0	1
gensandbox_username	224	164	60	1	0	1
gensandbox_path	220	160	60	1	0	1
gensandbox_common_names	3	0	3	1	0	1
gensandbox_drive_size	6	2	4	1	0	1
gensandbox_drive_size2	5	2	3	1	0	1
gensandbox_sleep_patched	2	1	1	1	0	1
gensandbox_one_cpu	2	1	1	2	1	1
gensandbox_one_cpu_GetSystemInfo	2	1	1	1	0	1
gensandbox_less_than_onegb	3	1	2	1	0	1
gensandbox_uptime	2	1	1	1	0	1
gensandbox_IsNativeVhdBoot	1	0	1	1	0	1
hooks.c						
check_hook_DeleteFileW_m1	1	0	1	1	0	1
check_hook_ShellExecuteExW_m1	1	0	1	1	0	1
check_hook_CreateProcessA_m1	1	0	1	1	0	1

continued ...

. . . continued

Module name	unaided			extended		
Check function	Tot	T	F	Tot	T	F
vbox.c						
vbox_reg_key1	244	120	124	1	0	1
vbox_reg_key2	244	120	124	1	0	1
vbox_reg_key3	4	2	2	1	0	1
vbox_reg_key4	244	120	124	1	0	1
vbox_reg_key5	4	2	2	1	0	1
vbox_reg_key7	4	2	2	1	0	1
vbox_reg_key8	4	2	2	1	0	1
vbox_reg_key9	∄	-	-	1	0	1
vbox_reg_key10	244	120	124	1	0	1
vbox_sysfile1	496	480	16	1	0	1
vbox_sysfile2	∄	-	-	1	0	1
vbox_mac	∞	-	-	1	0	1
vbox_devices	31	30	1	1	0	1
vbox_traywindow	3	2	1	1	0	1
vbox_network_share	3	1	2	1	0	1
vbox_processes	∞	-	-	1	0	1
vbox_wmi_devices	3	0	3	3	0	3
vmware.c						
vmware_reg_key1	∄	-	-	1	0	1
vmware_reg_key2	4	2	2	1	0	1
vmware_sysfile1	4	2	2	1	0	1
vmware_sysfile2	4	2	2	1	0	1
vmware_mac	∞	-	-	1	0	1
vmware_adapter_name	∞	-	-	1	0	1
vmware_devices	7	6	1	1	0	1
vmware_wmi_serial	3	0	3	3	0	3

continued . . .

... continued

Module name	unaided			extended		
Check function	Tot	T	F	Tot	T	F
qemu.c						
`qemu_reg_key1`	244	120	124	1	0	1
`qemu_reg_key2`	244	120	124	1	0	1
`qemu_cpu_name`	1	0	1	1	0	1
bochs.c						
`bochs_reg_key1`	244	120	124	1	0	1
`bochs_cpu_amd1`	1	0	1	1	0	1
`bochs_cpu_amd2`	1	0	1	1	0	1
`bochs_cpu_intel1`	1	0	1	1	0	1
sandboxie.c						
`sboxie_detect_sbiedll`	1	0	1	1	0	1
cuckoo.c						
`cuckoo_check_tls`	4	1	3	4	1	3
wine.c						
`wine_detect_get_unix_file_name`	1	0	1	1	0	1
`wine_reg_key1`	4	2	2	1	0	1

$\not{T}$: Timed-out, i.e. the exploration could have terminated (thanks to angr heuristics) but it was forcibly interrupted after running for more than 1 hour and/or thrashing the machine.

∞: Unbounded, i.e. because of the particular iterative nature of the check, the exploration would never have terminated.

4.1.2 Remarks

Looking at the results, in conclusion, should prove beyond any reasonable doubt the necessity for properly handling evasive detective checks in symbolic execution. Unaided symbolic execution of the entire pafish binary would predictably not have been tractable. In fact, since the checks are all unconditionally performed one after the other (independently of their result), the total number of generated paths would be the product of the total number of paths generated by each check

(i.e. the product of the entire *unaided-total* column). However, and this is what is indeed striking, even the conditional execution of just some of these checks could prove to be a grave impediment to the exploration. Consider for example the very common case of checks based on string comparison, like most of the regkey and file checks: even with the hard bounds introduced by angr's default SimProcedures for the string handling functions, they generated more than 200 paths per single artifact. Multi-artifact checks like `vmware_reg_key1`, or `vbox_sysfile2` actually thrashed the machine we were conducting tests on. Finally, the tests marked with ∞ in the table, because of their particular iterative nature (e.g. traversing a symbolic linked list), could singlehandedly compromise the feasibility of the symbolic exploration.

Our extension was able to perfectly handle most of the checks. Few of them required minor interventions, e.g. adding a particular keyword to the blacklist, or a new key to the dictionary of sensitive registry keys. Others instead relied on particular accessory functions (e.g. *libc* `toupper`, or `snprintf`), for which angr at the moment does not have a corresponding useful SimProcedure; thanks to the DLL loading however, all we had to do was unhook these few functions in order to use the actual Win32 implementation code present in the corresponding loaded module. Finally, it is worth noting that, though the available source code helped in the testing process, it was primarily the detailed logging we built in the implemented patches that made all the necessary modifications extremely easy to pinpoint and implement.

4.2 Malware

After the extensive testing with the pafish demonstration tool, we set out to evaluate the extension on real malware samples. To find suitable specimens on which to test the project, we filtered several virus collections, mainly on the basis of analysis-related string presence and sample size. Given the practical limitations of symbolic execution, it was fundamental to select malware samples adequate for testing purposes. That is samples of real malware, but of relatively small size, and preferably not employing heavy obfuscation techniques that would unnecessarily complicate the testing process (being practically orthogonal to our purposes).

We requested and obtained access to the VirusShare [28] malware repository. The site has a torrent tracker serving up very large packages of malware samples (dozens of GBs each) ordered by collection date. We downloaded several packages,

and sifted through them using simple bash shell scripts, looking for string content suggesting the presence of anti-analysis checks (e.g. registry keys or file system artifacts).

We also requested and obtained access to VirusTotal [29] private APIs, that would allow us to download selected malware samples from the site. VirusTotal was kind enough to also grant us the use of their *Intelligence* platform, allowing finer-grained filtering of their whole dataset. The Intelligence platform also allows for batch download of the files matching a given query, so we could further filter the downloaded samples locally using the previously mentioned bash scripts. Additionally, we also made use of the Intelligence platform *Malware hunting* service, that allowed us to hook onto the live stream of files submitted to VirusTotal and get notified whenever one of them matched a certain *rule* written in the YARA language [32] (i.e. a description of textual or binary patterns present in the sample).

All gathered samples were then briefly analyzed in greater detail using conventional static and dynamic analysis tools, to further ascertain their characteristics and capabilities.

After a long screening process, we finally selected the *Kasidet* backdoor malware family as the most suitable for testing the developed framework. Variants of this family date as far back as 2013 and have successfully compromised thousands of computers across more than 50 different countries. The malware is also known under the alias of *Neutrino Bot*, and its instances typically contain unique identifiers to keep track of which computers have been compromised by each campaign. The backdoor gathers information on the infected computer, and communicates with a command-and-control (C&C) server. It installs itself in a number of executable files in the `%APPDATA%` user folder, and modifies the registry so that it is automatically launched at every startup, ensuring persistence. Once activated, the malware allows a remote attacker to take over the infected machine by exchanging files with the server and executing shell commands. Several variants also possess additional information-stealing abilities, like key-logging, browser-hooking and memory-scraping capabilities.

This malware family was selected because most of the samples contain many analysis detection mechanisms, ranging from debugger detection to anti-virtualization and anti-sandbox checks. Additionally, many of the available samples do not employ packing or heavy obfuscation techniques, making them perfect candidates for our testing purposes.

4.2.1 Evaluation

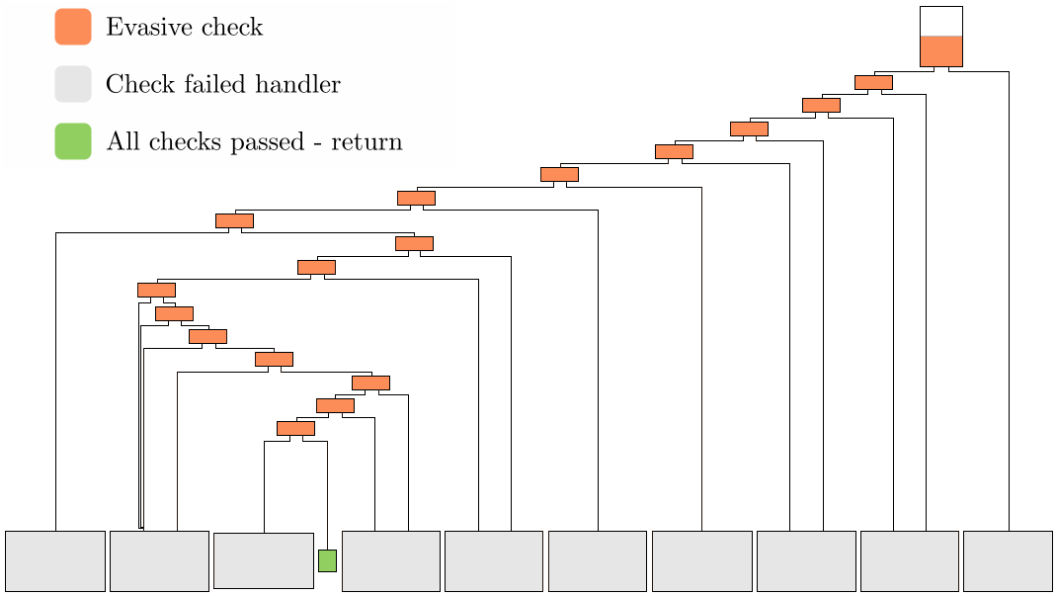
The particular sample we will analyze in this write-up was downloaded from the VirusTotal database (MD5 signature is `de1af0e97e94859d372be7fcf3a5daa5`). The sample performs anti-analysis activities right at the beginning of its execution: in the `WinMain` entry point (`0x407916`), just after initializing the `Winsock` library, a dedicated evasive function is called at address `0x407967`. The function, starting at address `0x4020F4`, performs checks in a conditional cascading fashion: if a check fails, a message box showing an error is displayed and execution terminates with a concealed invocation of `ExitProcess`; otherwise control flows to the next check. Each check is, in turn, encapsulated in a function returning `1` if the check passed (i.e. no detection), `0` otherwise. In order, the malware performs the following checks:

- Sleep patching check
- Debugger checks
- User name checks
- File path checks
- Wine emulation check
- Injected DLLs check
- VMware virtualization checks
- VirtualBox virtualization checks
- QEMU emulation checks
- Drive size check

A high-level picture of the control flow graph of the evasive function, automatically generated by the IDA disassembler [21], is displayed in Figure 4.1. There is a total of 17 different check functions, many of them actually performing more than one check. It should be evident already the path explosion problem we would have by executing the section in a purely symbolic fashion. Additionally, though this particular sample has all the evasive functionalities nicely encapsulated in a single method that we may decide to simply bypass completely, it should be clear that the same checks could have been peppered through the entire initialization section of the binary, making their individual bypass, not to mention their discovery, much more difficult and time consuming.

In the following sections we will have a more in depth look at the various checks employed by the malware.

Figure 4.1. Control flow graph of the evasive check calling function



Sleep-patched check

Starting at address `0x402AE5`, we have a standard `Sleep`-patched check. It calls `GetTickCount`, sleeps for 500 ms, then calls `GetTickCount` again, thus computing the elapsed time: if it is less than 450 ms then a clear time manipulation is detected and it returns 0; otherwise the check passes returning 1.

```
sleep_patch_check proc near
  push    esi
  mov     esi, ds:GetTickCount
  push    edi
  call    esi ; GetTickCount
  push    500 ; dwMilliseconds
  mov     edi, eax
  call    ds:Sleep
  call    esi ; GetTickCount
  sub     eax, edi
  cmp     eax, 450
  pop     edi
  setnl   al
  pop     esi
  retn
sleep_patch_check endp
```

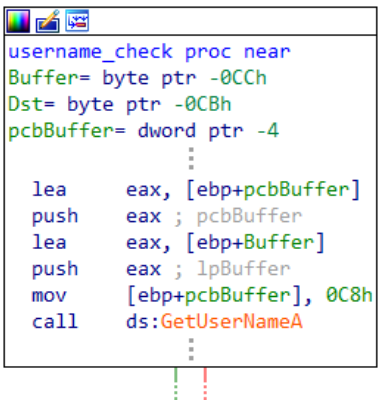
Our extension handles this check easily with the aid of the *paranoid* plugin, as explained in section 3.3.5. No additional intervention was required.

Debugger checks

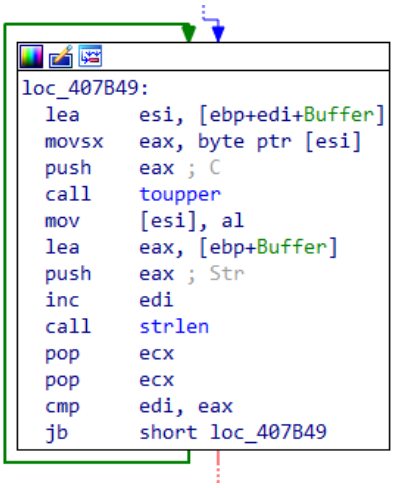
The next two checking functions perform simple debugging detection. The first, at `0x403D7F`, uses Windows `IsDebuggerPresent` API, while the second, at `0x403D8B`, invokes the `CheckRemoteDebuggerPresent` API. These behavioral patterns were both modeled and patched in our extension as described in section 3.3.2, so no addition was necessary.

User name check

Next, at address `0x407AF6`, we have a long subroutine that compares the machine user name against presumably standard and/or fingerprinted names.



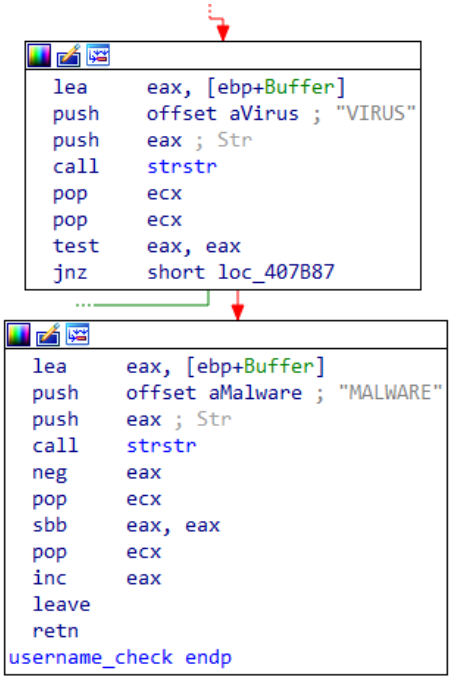
It calls Windows `GetUserNameA` API to retrieve the name of the user associated with the current process.



It then converts the returned name to uppercase, iteratively invoking libc `toupper` for each character.

Finally, libc `strstr` function is repeatedly used to perform the comparison with each of the following names:

- `"MALTEST"`
- `"TEQUILABOOMBOOM"`
- `"SANDBOX"`
- `"VIRUS"`
- `"MALWARE"`



In our extension, we bypass this kind of check by patching `GetUserName` with a SimProcedure simply returning `"John"`, a concrete innocuous name (as described in 3.3.1).

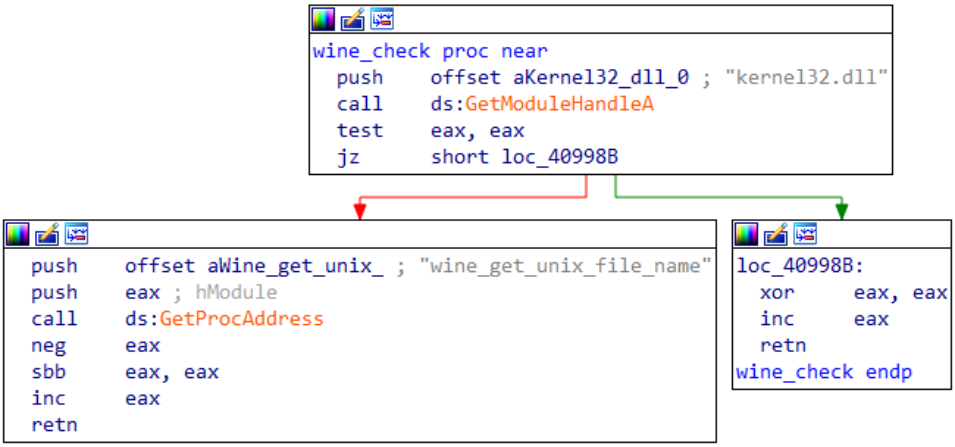
File path checks

The function at `0x407BEA`, structurally very similar to the previous one, looks instead at the executable file path, checking for the presence of popular sandbox mount points or directories. To do this, it invokes the `GetModuleFileNameA` API, and then uses libc `toupper` and `strstr` to perform a case-insensitive comparison with the following strings:

- `"\\SAMPLE"`
- `"\\VIRUS"`
- `"SANDBOX"`

We already modeled and patched this kind of check in our extension exactly as described in 3.3.3, essentially returning a concrete innocuous file path. No additional intervention was thus necessary.

Wine emulation check



Next, the function at `0x40996A` performs a simple but reliable check for the presence of the Wine compatibility layer. It gets a handle to the `kernel32.dll` module with the `GetModuleHandleA` API, and then invokes `GetProcAddress` on it to check for the presence of the injected function `wine_get_unix_file_name`.

While we did not model and patch this kind of behavior explicitly in our extension, since we load all required DLLs from a vanilla Windows XP installation,

the verification is actually concretely performed (as explained for hooks detection in 3.3.2) by both `GetModuleHandle` and `GetProcAddress`, and thus the check automatically passes.

Injected DLLs check

The subroutine starting at address `0x407FC2` then checks for the presence of several DLLs injected by some common analysis-related products. It repeatedly invokes the `GetModuleHandleA` API with the following targets:

- `"sbiedll.dll"` (Sandboxie)
- `"dbghelp.dll"` (VMware)
- `"api_log.dll"` (SunBelt SandBox)
- `"dir_watch.dll"` (SunBelt SandBox)
- `"pstorec.dll"` (SunBelt Sandbox)
- `"vmcheck.dll"` (Windows Virtual PC)
- `"wpespy.dll"` (Winsock Packet Editor Pro)

As explained in 3.3.2, since we load all (and only) the required DLLs along with the executable, `GetModuleHandleA` performs a concrete verification and thus all checks pass automatically.

Registry key artifacts checks

The next eight functions all perform registry key artifacts checks to detect various analysis-related products.

VMware virtualization checks The first two functions try to detect whether the executable is being run on a VMware virtual machine by checking for registry artifacts.
The first one, at address `0x40989F`, performs a registry key *value* check: it invokes `RegOpenKeyExA` to open the registry key `HARDWARE\DEVICEMAP\Scsi\Scsi Port 0\Scsi Bus 0\Target Id 0\Logical Unit Id 0`, queries the value of its `Identifier` field with `RegQueryValueExA`, and finally performs a case-insensitive comparison with the string `"VMWARE"`.
The second one, at address `0x409943`, instead performs a simpler registry key *presence* check, invoking `RegOpenKeyExA` with target `SOFTWARE\VMware, Inc.\VMware Tools`.

VirtualBox virtualization checks The next four functions check registry artifacts to detect VirtualBox virtualization services instead.
The first subroutine, at address `0x40968C`, performs the same registry key *value* checking on the `Identifier` field of the `HARDWARE\DEVICEMAP\Scsi\Scsi Port 0\Scsi Bus 0\Target Id 0\Logical Unit Id 0` key performed by the first anti-VMware check, this time, however, comparing the queried value with the string `"VBOX"`.
The second function (`0x409730`) performs another registry key *value* check, this time on the `SystemBiosVersion` field of the `HARDWARE\Description\System` key, comparing it with the string `"VBOX"`.
The third function (`0x4097D4`) performs a simpler registry key *presence* check on key `SOFTWARE\Oracle\VirtualBox Guest Additions`.
Finally, the fourth subroutine, at address `0x4097FB` performs another regkey *value* check on the `VideoBiosVersion` field of the `HARDWARE\Description\System` key, looking for the string `"VIRTUALBOX"`.

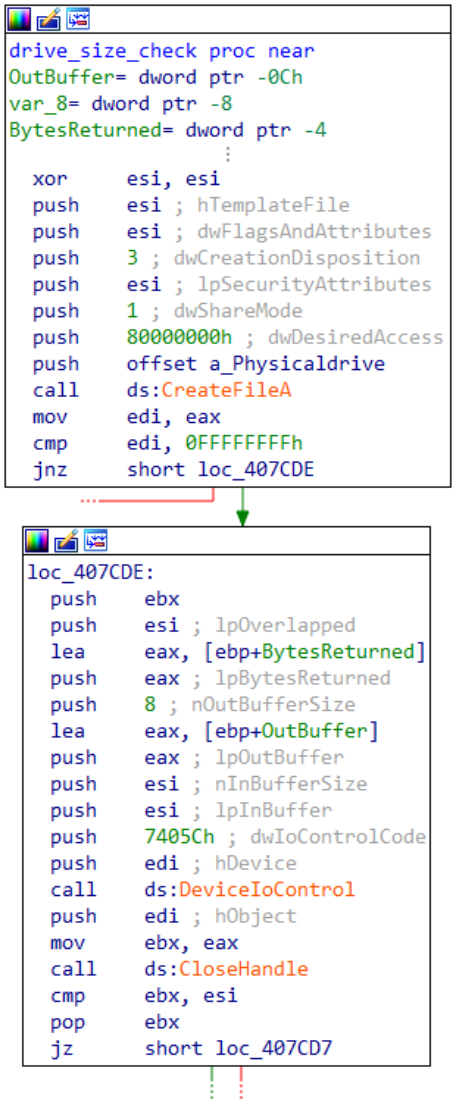
QEMU emulation checks At addresses `0x407D26` and `0x407DCA` respectively, we have the final two functions performing registry key *value* checks to detect emulation under QEMU.
The first one checks the `Identifier` field of the `HARDWARE\DEVICEMAP\Scsi\Scsi Port 0\Scsi Bus 0\Target Id 0\Logical Unit Id 0` key, while the second one queries the `SystemBiosVersion` field of the `HARDWARE\Description\System` key. In both cases the returned value is compared with the string `QEMU`.

Both of the registry key checking patterns - *presence* checking through `RegOpenKeyExA`, and *value* checking through `RegOpenKeyExA` and `RegQueryValueExA` - were modeled and patched in our extension as described in section 3.3.1. Additionally, all keys were previously encountered in our testing with Paranoid Fish.

Drive size check

Finally, the last function (`0x407CA8`) performs an overly modest drive size check. Win32 `CreateFileA` is invoked to get a handle to the main hard disk drive (through the special/device file identifier `\\.\PhysicalDrive0`). The returned handle is then passed to the `DeviceIOControl` API along with the control code `0x7405C` (i.e `IOCTL_DISK_GET_LENGTH_INFO`) to get the disk size in bytes. The value is then divided by 1073741824 (i.e. 2^{30}) to obtain the size in GBs and the final comparison is performed: if the drive size is greater than 10 GBs the check passes returning 1, otherwise it fails returning 0.

This kind of overly modest hardware check was modeled and patched in our extension as described in section 3.3.4, essentially concretizing the returned value with a safe, large enough size of 256 GBs. No additional intervention was thus necessary.



4.2.2 Remarks

The evasive checks calling function when run unaided thrashed the testing machine after approximately one hour. Aided by the developed extension instead, the exploration of the entire function generated exactly one path terminating in a successful return in less than 10 seconds.

Our extension was able to perfectly handle all the checks. Practically the only nuisance we had was with the case-insensitive comparisons, due to angr's lack of a SimProcedure for libc `toupper` function. Using the actual implementation of the function from the `msvcrt.dll` module, unfortunately, resulted in a baffling symbolic Instruction Pointer (IP) that stalled the exploration. However, we resolved quite easily by writing our own SimProcedure for the function (that was later also merged into the angr project on GitHub). Thanks to the detailed

logging built in the developed patches, the few necessary interventions were very easy to locate and implement.

4.3 Conclusions

In this chapter we have reported the results obtained by the experimental evaluation of the Angr extension we developed.

Both the tests on Paranoid Fish and the malware samples convinced us of the danger that malware evasion poses to the application of symbolic execution. Especially severe was, as expected, the impact of inherently iterative checks, like the extremely common ones based on string comparisons.

The same tests, however, convinced us of the practicality of the developed extension. The patches introduced in 3.3 were enough to cover more than 90% of the techniques we encountered, allowing us to completely bypass the symbolic exploration of these checks with practically no major alteration. Finally, the detailed built-in logging and the simplicity of the extension design made the few necessary additions extremely easy to pinpoint and implement.

Chapter 5

Conclusions

Malware analysis technologies are constantly being researched and improved to confront the ever evolving attacks of the staggering amount of malicious software samples surfacing every day. Symbolic execution techniques have been recently gaining traction in the security domain, and several academic papers have begun to inquire about its applicability to malware analysis. Symbolic evaluation blurs the line between static and dynamic analysis techniques. Its strength lies in the ability to potentially cover all possible execution paths, identifying for each the concrete input values that would materialize it. This power actually makes it a very promising tool for the study of trigger-based behaviors extremely widespread in malicious software. Unfortunately, this power also comes at a rather obvious price: a fundamental weakness of symbolic execution is the exponential growth of the number of generated paths with respect to the program size, generally known as the *path explosion* problem. Detective malware evasion techniques, which are inherently conditional, then represent an even graver hindrance to the applicability of symbolic execution, since they correspond to critical branching points: at least half of the paths generated by the exploration of a single evasive check do not lead to any interesting malicious behavior and are thus completely irrelevant from an analysis standpoint.

Motivated by this observation, in this thesis we performed a meticulous profiling of the most widespread evasive checks, identifying their key behavioral patterns in terms of Windows APIs interactions, and we built an extension for the Angr binary analysis and symbolic execution framework that is able to constrain the symbolic evaluation so as to automatically pass these checks. The developed extension appears to be an extremely practical solution to the aforementioned problem, since it has the potential to completely bypass the symbolic exploration

of the evasive checks, and can be easily integrated with any other ANGR-based project.

The tests conducted on the Paranoid Fish demonstration tool and on real malware samples, finally, reinforced our observations about the severity of the threat that evasion poses for symbolic execution, while encouragingly confirming the effectiveness of our proposed solution.

5.1 Future developments

In this thesis the focus has been on improving the applicability of symbolic execution to the malware domain by patching evasion techniques. The work, in other terms, serves as a basis, a support for any future project conjugating malware analysis and symbolic evaluation.

In addition to being an aid to future works, however, we feel our extension also has the potential to be developed further. A simple addition to implement would be, for example, the ability to generate automatic reports on the applied patches. This would really just require a re-purposing of the extensive logging abilities already present in the extension. Such a report would then indicate the machine configuration necessary to concretely generate the corresponding path in a normal execution of the malware. It would therefore represent an immediate transfer of the information gathered through the symbolic exploration to more standard malware analysis methodologies, enriching them. We can imagine it being parsed by an automated script in order to automatically reproduce the configuration on a sandbox before performing the usual dynamic analysis techniques.

A more involved but very interesting direction to investigate with the aid of symbolic execution would be the shift from *targeted patching* to a more general *abstract detection* of the evasive measures employed by a malware sample. ANGR is a simple yet quite capable binary analysis framework and, borrowing from the extensive CFG exploration literature, it could theoretically be possible to define ad hoc analyses and heuristics to identify critical branching points possibly corresponding to evasive checks. Such an approach could, again theoretically, exploit the exhaustiveness of symbolic exploration in order to detect evasive attempts, and could therefore represent a promising alternative anti-evasion technique.

Appendices

Appendix A

Using the extension

In this appendix we briefly describe how to install and use the Angr extension we developed. In the following description we are going to assume the reader has basic familiarity with Python and Angr, and both are installed on his machine. For Angr we recommend the official documentation at `https://docs.angr.io/`.

A.1 Installation

The extension is hosted and available on GitHub at `https://github.com/fabros/angr-antievasion`, and can thus be downloaded or cloned in the usual ways. Once obtained, all that is necessary to do is to add the corresponding directory to the `PYTHONPATH` so that we may import the extension in our angr scripts.

A.2 Package structure

The extension is contained in the `angr_antievasion` package. The package automatically exposes all underlying functionalities thanks to the `__init__.py` file. `stdcall_simproc.py` and `paranoid_plugin.py` respectively contain the definitions of `StdcallSimProcedure` and `ParanoidPlugin` we described in 3.2.1.
`win32_patches.py` contains the Win32 API models, while `cpu_instr_patches.py` contains the custom *VEX dirty helpers*, all of which we use in order to patch the symbolic execution of evasive checks, as described in section 3.3.
Finally, `utils.py` contains several utilities and auxiliary functions that act as the preferred interface to apply the extension. In particular the function `hook_all(project)` applies all the available patches to the specified angr project.

A.3 Windows DLLs importing

When analyzing a PE file (i.e. a Windows binary) with Angr we strongly recommend to load all its required DLLs along with it.

The first and most involved reason is linked with symbolic stubs calling conventions. The library functions and definitions - collectively known as *symbols* - required by a PE file are listed in the PE header (in particular in the Import Address Table, or IAT). During the initial loading of a PE file, angr analyzes the header to perform the necessary symbol analysis and resolution. Symbols that cannot be resolved to the corresponding implementations - e.g., because the required DLL was not loaded - are mapped by angr on a fictitious `ExternObject`, an empty memory region whose sole purpose is to allow angr to hook the symbols. Then angr hooks each of the fictitiously resolved symbols either with an ad hoc SimProcedure for it (if present in the available SimLibraries), or with a simple `ReturnUnconstrained` stub. The problem when hooking a *fictitiously resolved symbol* with a stub is that, since no information is available on it, angr defaults to the x86 architecture standard calling convention, i.e. *cdecl* (caller-cleanup). However, Windows standard calling convention is *stdcall* (callee-cleanup). Unfortunately a simple change of angr's settings is not advisable, since not all of Windows APIs actually use *stdcall* (e.g., modules like the `msvcrt.dll` that implement the C standard library functions actually use the *cdecl* calling convention). The best thing to do in our experience is to just load all the required DLLs along with the binary. This would completely prevent the need for the `ExternObject` and the `ReturnUnconstrained` stubs. Additionally, should we still want to stub out a symbol (for example to avoid the symbolic exploration overhead we would incur in when executing the actual implementation code) we can reliably do so either individually or using the available angr's analyses (e.g., the `CalleeCleanupFinder` analysis that can identify and stub all calee-cleanup functions in a binary).

The second simpler reason why we recommend to load all required DLLs is actually linked to evasive techniques. In order to effectively bypass many hooks and DLLs injection checks, the best solution, as we explained in 3.3.2, is to actually provide angr with all the modules required by the sample so that the checks can be concretely carried out.

Angr is able to automatically use the libraries of the system it is installed on. Since we're targeting PE files (i.e. Windows executables), if we were operating

Angr from a Windows machine, all the right DLLs would be ideally loaded automatically. Unfortunately, at the time of this writing, Angr support is practically exclusive to Linux systems. Operating from a Linux machine, we instead need to explicitly provide Windows DLLs to our angr script. Angr, upon creating a project, tries to import all and only the DLLs required from the binary under analysis. The easiest way to proceed then is to copy all DLLs from the `System32` folder of a genuine Windows installation [1] into a local directory. Each time we need to analyze a PE file then we can simply point the CLE loader to the directory when loading the binary. Look at the Python snippet below for how to do just that.

```
project = angr.Project(
  './sample.exe',  # path to the main binary
  load_options={
    'auto_load_libs': True,
    'use_system_libs': False,
    'case_insensitive': True,  # Windows DLL resolution is case-insensitive
    'custom_ld_path': './windows_dlls',  # path to the DLL directory
    'except_missing_libs': True,
  }
)
```

The snippet configures the project to load all the required DLLs from the specified directory and to actually throw an exception if any of them cannot be found.

A.4 Usage example snippet

A sample usage snippet of our extension in an angr script is reported below.

```
#!/usr/bin/python

import angr
import angr_antievasion

project = angr.Project(
```

[1] In relatively recent Windows versions (i.e. post-XP) be careful to copy *all* DLL files from the folder. Some are actually hidden and the standard `cp` command is not enough. Use `robocopy` instead.

```
  './sample.exe',  # path to the main binary
  load_options={
    'auto_load_libs': True,
    'use_system_libs': False,
    'case_insensitive': True,  # Windows DLL resolution is case-insensitive
    'custom_ld_path': './windows_dlls',  # path to the DLL directory
    'except_missing_libs': True,
  }
)

# stub out all imports (if we want to)
project.analyses.CalleeCleanupFinder(hook_all=True)

# apply all the patches in the extension
angr_antievasion.hook_all(project)

# perform the analysis...
```

Complete, more involved usage examples can be found in the `testing` directory of the project that contains both the pafish and malware test scripts for the evaluation we described in Chapter 4.

A.5 Altering or expanding the extension

As we saw in Chapter 4, the implemented patches we described in 3.3 were enough to cover more than 90% of the checks we encountered. However, we designed the extension to be as easy to modify and expand as possible so as to facilitate the implementation of any necessary alteration.

The recommended procedure would be to fork the project on GitHub and alter the extension as needed. In particular, handling practically any type of checking pattern would very likely just require either a modification of the currently available Win32 API models or the addition of new ones. In the former case, it would be sufficient to simply modify the relevant model, extending or modifying its logic as necessary. In the latter case, we would recommend defining the new SimProcedure in the `win32_patches.py` file, carefully matching its name with the API symbol it is supposed to hook. This way, the previously mentioned `hook_all` function in the `utils.py` file would then be able to automatically hook the corresponding symbol with the newly provided SimProcedure without any additional intervention.

Finally, do consider submitting a pull request to merge your changes with the original repository. We would be incredibly happy to enrich the project with your contribution.

Bibliography

[1] D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin, "Bitscope: Automatically dissecting malicious binaries", 2007.

[2] A. Moser, C. Kruegel, and E. Kirda, "Exploring Multiple Execution Paths for Malware Analysis", in *2007 IEEE Symposium on Security and Privacy (SP '07)*, IEEE, May 2007. DOI: 10.1109/sp.2007.17.

[3] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna, "Efficient detection of split personalities in malware", in *In Proc. of the Symposium on Network and Distributed System Security (NDSS*, 2010.

[4] D. Kirat, G. Vigna, and C. Kruegel, "BareBox: efficient malware analysis on bare-metal", in *Proceedings of the 27th Annual Computer Security Applications Conference on - ACSAC '11*, ACM Press, 2011. DOI: 10.1145/2076732.2076790.

[5] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox Fuzzing for Security Testing", *Queue*, vol. 10, no. 1, p. 20, Jan. 2012. DOI: 10.1145/2090147.2094081.

[6] D. Kirat, G. Vigna, and C. Kruegel, "Barecloud: Bare-metal Analysis-based Evasive Malware Detection", in *Proceedings of the 23rd USENIX Conference on Security Symposium*, USENIX Association, 2014, ISBN: 978-1-931971-15-7. [Online]. Available: http://dl.acm.org/citation.cfm?id=2671225.2671244.

[7] D. Kirat and G. Vigna, "MalGene: Automatic Extraction of Malware Analysis Evasion Signature", in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*, ACM Press, 2015. DOI: 10.1145/2810103.2813642.

[8] F. Zhang, K. Leach, A. Stavrou, H. Wang, and K. Sun, "Using Hardware Features for Increased Debugging Transparency", in *2015 IEEE Symposium on Security and Privacy*, IEEE, May 2015. DOI: 10.1109/sp.2015.11.

[9] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques", *CoRR*, vol. abs/1610.00502, 2016. [Online]. Available: http://arxiv.org/abs/1610.00502.

[10] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis", in *IEEE Symposium on Security and Privacy*, 2016.

[11] R. Baldoni, E. Coppa, D. C. D'Elia, and C. Demetrescu, "Assisting Malware Analysis with Symbolic Execution: A Case Study", in *Lecture Notes in Computer Science*, Springer International Publishing, 2017, pp. 171–188. DOI: 10.1007/978-3-319-60080-2_12.

[12] *angr, a binary analysis framework*, Accessed on Thu, October 05, 2017. [Online]. Available: http://angr.io/.

[13] *angr - GitHub*, Accessed on Thu, October 05, 2017. [Online]. Available: https://github.com/angr/.

[14] *Avast*, Accessed on Thu, October 05, 2017. [Online]. Available: https://www.avast.com/.

[15] *AV-TEST – The Independent IT-Security Institute*, Accessed on Thu, October 05, 2017. [Online]. Available: https://www.av-test.org/en/statistics/malware/.

[16] *bochs: The Open Source IA-32 Emulation Project*, Accessed on Thu, October 05, 2017. [Online]. Available: http://bochs.sourceforge.net/.

[17] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, "Automatically Identifying Trigger-based Behavior in Malware", in *Botnet Detection*, Springer US, pp. 65–88. DOI: 10.1007/978-0-387-68768-1_4.

[18] *Cuckoo Sandbox*, Accessed on Thu, October 05, 2017. [Online]. Available: https://www.cuckoosandbox.org/.

[19] *Evasive Malware's Gone Mainstream - Lastline*, Accessed on Thu, October 05, 2017. [Online]. Available: https://www.lastline.com/labsblog/labs-report-at-rsa-evasive-malwares-gone-mainstream/.

[20] *Explained: Packer, Crypter, and Protector - Malwarebytes Labs*, Accessed on Thu, October 05, 2017. [Online]. Available: https://blog.malwarebytes.com/cybercrime/malware/2017/03/explained-packer-crypter-and-protector/.

[21] *Hex-Rays IDA*, Accessed on Thu, October 05, 2017. [Online]. Available: https://www.hex-rays.com/products/ida/.

[22] *How To Build An Effective Malware Analysis Sandbox - Lastline*, Accessed on Thu, October 05, 2017. [Online]. Available: https://www.lastline.com/labsblog/different-sandboxing-techniques-to-detect-advanced-malware/.

[23] *Obfuscation: Malware's best friend - Malwarebytes Labs*, Accessed on Thu, October 05, 2017. [Online]. Available: https://blog.malwarebytes.com/threat-analysis/2013/03/obfuscation-malwares-best-friend/.

[24] *Oracle VM VirtualBox*, Accessed on Thu, October 05, 2017. [Online]. Available: https://www.virtualbox.org/.

[25] *Paranoid Fish*, Accessed on Thu, October 05, 2017. [Online]. Available: https://github.com/a0rtega/pafish/.

[26] *QEMU*, Accessed on Thu, October 05, 2017. [Online]. Available: https://www.qemu.org/.

[27] *Sandboxie*, Accessed on Thu, October 05, 2017. [Online]. Available: https://www.sandboxie.com/.

[28] *VirusShare*, Accessed on Thu, October 05, 2017. [Online]. Available: https://virusshare.com/.

[29] *VirusTotal*, Accessed on Thu, October 05, 2017. [Online]. Available: https://www.virustotal.com/.

[30] *VMware Virtualization*, Accessed on Thu, October 05, 2017. [Online]. Available: https://www.vmware.com/.

[31] *WineHQ*, Accessed on Thu, October 05, 2017. [Online]. Available: `https://www.winehq.org/`.

[32] *YARA - The pattern matching swiss knife for malware researchers*, Accessed on Thu, October 05, 2017. [Online]. Available: `https://virustotal.github.io/yara/`.