Behavior-Driven Development in Malware Analysis

Thomas Barabosch, Elmar Gerhards-Padilla
Fraunhofer FKIE
Bonn, Germany
firstname.lastname@fkie.fraunhofer.de

This paper was presented at Botconf 2015, Paris, 2-4 December 2015, www.botconf.eu
It is published in the Journal on Cybercrime & Digital Investigations by CECyF, https://journal.cecyf.fr/ojs

DOI: 10.18464/cybin.v1i1.9

***Abstract*—A daily task of malware analysts is the extraction of behaviors from malicious binaries. Such behaviors include domain generation algorithms, cryptographic algorithms or deinstallation routines. Ideally, this tedious task should be automated. So far scientific solutions have not gotten beyond proof-of-concepts. Malware analysts continue to reimplement behaviors of interest manually. However, often times they merely translate the malicious binary assembler code to a higher-level language. This yields to poorly readable and undocumented code whose correctness is not ensured. Furthermore, the current process that malware analysts are following leads to a suboptimal focusing since they deal with too much binary code at once. In this paper, we aim at overcoming these shortcomings by improving the malware analysis process regarding the reimplementation of malicious behaviors. We achieve this by integrating Behavior-Driven Development in the malware analysis process. We explain in detail how the integration of Behavior-Driven Development into the malware analysis process can be done. In a case study on the highly obfuscated malware Nymaim, we show the feasibility of our approach.**

I. INTRODUCTION

In 2015, profound malware analysis continues to be a highly manual task. Cyber criminals employ, for example, code obfuscation or anti-debugging techniques just with the objective of stealing malware analysts' precious time. Even though the research community keeps working hard on these problems, many promising approaches never reach marketability or are never released publicly. Take for example the extraction of behaviors from malicious binaries. Kolbitsch et al. proposed a system that extract behaviors automatically. Their system, however, was never publicly shared with the malware research community. Furthermore, there existed several limitations such as multi-threading or obfuscation that restrict this approach to some malware families. To overcome these limitations one would have to stitch their system together with other approaches that, for example, deobfuscate malicious binaries. To solve this problem in practice, a tremendous engineering effort would be necessary. But still there are open questions like how to detect zero-day anti-analysis techniques and how to circumvent them automatically?

However, each day malware analysts are facing the reimplementation task, i.e. reimplementing the underlying algorithms of a malicious behavior. For example, this could be a domain generation algorithm (DGA) in order to anticipate future domain names or this could be a cryptographic algorithm in order to decrypt automatically configuration files of a malware family. Since the aforementioned scientific approaches are rather proof-of-concepts, practitioners are extracting algorithms from malicious software still manually. Often times they are merely translating from the binary's assembler code to a higher-level language. This leads to several problems. Among these problems are incorrect, poorly readable and undocumented code. Depending on further usage of extracted behaviors, these problems might lead to trouble among the team or even waste of money in case incorrect DGAs are used for automatically registering domains. Furthermore, the sheer mass of binary code that belongs to a typical behavior yields to a suboptimal focusing on too much code at once. We believe that malware analysts could be more efficient when using an improved process for tackling the reimplementation task.

In this paper, we aim at overcoming these problems by integrating Behavior-Driven Development into the malware analysis process. Our approach improves the way malware analysts tackle the reimplementation task. We achieve this by combining malware analysis with a testing-driven process. Rigorous testing is a fundamental part of today's software development processes. Many industrial case studies (e.g. [1] or [2]) state that testing-driven processes come with a lot of benefits such as a significantly reduced defect rate. By enhancing malware analysis with a testing-driven process, we overcome the aforementioned shortcomings of the current state of the practice such as incorrect or complex code. We chose Behavior-Driven Development (BDD) as testing process. Since this process, allows malware analysts to describe their observations in natural language. Furthermore, BDD rests on the Hoare logic that is used for proofing (partial) correctness of programs. This gives it a profound theoretic background. We show the feasibility of our approach in a detailed case study on the highly obfuscated malware family Nymaim. In this case study, we reimplement Nymaim's DGA with the help of BDD.

To summarize our contributions:

- **BDD in Malware Analysis.** We present a novel approach for tackling the reimplementation task by applying Behavior-Driven Development to malware analysis. This ensures correct (i.d. according to the system specification of the malware sample at hand), readable and documented code. Furthermore, it aids the reverse engineer in focusing on one piece of code at a time.
- **Case Study on Nymaim.** We show the feasibility of our process in a case study on the heavily obfuscated malware family Nymaim. We reimplemented its domain generation algorithm.

II. REIMPLEMENTATION TASK

Malware analysts facing frequently the task of extracting behaviors from malware samples. There are already solutions to this problem (e.g. [3] or [4]). However, these solutions fail to work in practice because of, for example, heavy obfuscation. While the research community is still working on these problems, practitioners have to deal with them each day. Since automatic extraction systems are still not applicable to real world malware, practitioners continue to analyze these behaviors of interest by hand and reimplement them in high-level languages like Python. In the following, we refer to this process as the *reimplementation task*. By reimplementation task, we mean the process of analyzing (malicious) binary code, understanding this code and delivering a reimplementation in a high-level programming language that works according to the system specification given by the binary.

In contrary to traditional software development where the customer provides the requirements, analysts have to derive the requirements from the malware sample by themselves. This means that the system specifications are entirely dictated by the malware sample at hand. As a consequence, care has to be taken such that bugs and non-standard behavior are also considered in the reimplementation.

Often times analysts just translate the behavior from assembly into another higher-level language. However, this leads to several problems. First, the code's functionality is not ensured. There are typically no tests that proof the code to work at least for some cases. This can have several consequences such as wrong reimplementations or non-confident malware analysts. Second, the code's readability is poor. Merely decompiling the compilation of a binary compiler yields to a totally different code structure that often times is not that elegant. This is comparable to translating with *Google Translate* (cf. Appendix A). Third, the code is not documented. Colleagues or third parties sometimes have to reverse engineer the reimplemented code in order to understand its meaning. Fourth, as a consequence of merely translating code from one language to another, it is not ensured that the underlying semantics of the code are 100% clear to the analyst (cf. Appendix A). Understanding of the code can, for example, lead to the detection of exploitable bugs. As a result of these four points, other colleagues or even third parties might run into problems when working with such a reimplementation.

This leads us to the problem that we are tackling in this paper: the current process how malware analysts reimplement behaviors found in malicous binaries. We think that a new process has to meet the following requirements: First, the solution should enable analysts to describe concisely and naturally what they observe. Hence, a description in natural language is required. The intuition here is that explaining observations in natural language forces the analyst to reflect them and it leads to a better understanding. Furthermore, this description in natural language serves as a documentation for colleagues or third parties. Second, the solution should continuously ensure that the code works. Therefore, it must be tested. Feedback loops should be very short and malware analysts should be confident that their code is working continuously. Third, the code should be concise and readable. This requires frequent refactorings of the code. Since we demand a suit of comprehensive tests, malware analysts do not have to fear refactorings. Unintended semantic changes that are introduced to the code at this stage would be directly detected due to the tests. Fourth, the process should improve the focusing of malware analysts by fading out regions of a behavior that are not important for the reimplementation of a submodule. All these requirements can be fulfilled when using a *-driven development process. In the next section, we discuss these processes in detail.

III. *-DRIVEN-DEVELOPMENT

In this section, we discuss software testing processes. At first, we have a look at *Software Testing* in general. This is followed by a discussion of two development processes that incorporate testing as core feature. The first process is *Test-Driven Development* (TDD) and the second process is *Behavior-Driven Development* (BDD).

A. Software Testing

The main objective of software testing is to test a software's functionally. It allows to find defects and failures. However, the input space is at least very large, if not infinite. In fact, the detection of all possible runtime errors of a software is undecidable. This can be proven by a reduction to the halting problem [5]. In addition to testing the functionality, non-functional requirements of a software can be tested. These include performance, scalability, usability and reliability.

However, problems arise when software testing is done infrequently. For example, in the waterfall model [6] testing/verification is a separate step after writing the software's code. In case the testing phase is not successful, the code has to be fixed in order to pass the tests. This leads to long debugging sessions of code that might have been written a couple of weeks ago.

B. Test-Driven Development

The consequence of the aforementioned problem was that software testing was integrated into the development process [7]. The main idea was writing a test first and then writing productive code in order to pass this test. From this idea the rule of three emerged: writing a test, making this test pass and refactoring the code. These steps are repeated until the software is implemented. This process comes with a lot of benefits, even though it might see tedious at first. These benefits include continuously working code, fast detection of bugs and no fear of refactorings.

Test are written in the form of unit tests. They test individual units of the software that might only be part of a certain behavior. The following listing in Python shows two simple unit tests of Python lists.

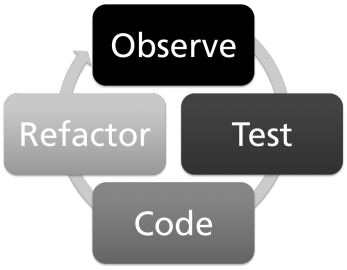
```
import unittest

class TestListMethods(unittest.TestCase):

  def test_empty_list_has_no_element(self):
    self.assertEqual(len([]), 0)

  def test_list_with_one_element(self):
    self.assertEqual(len([1]), 1)
```

Fig. 1: Overview of the BDD process applied to malware analysis.



These two tests test two fundamental characteristics of Python lists. The first test checks if the empty list does not contain any element. The second test tests if a list with one element has the length of one.

C. Behavior-Driven Development

Behavior-Driven Development (BDD) was introduced by North in 2006 [8]. BDD uses a domain specific language (DSL) in order to describe the behavior under test. It is furthermore strictly based on Hoare logic [9]. This logic allows to proof (partial) correctness of programs. The fundamental entity of this logic is the Hoare triple $\{P\}\ C\ \{Q\}$, where P and Q are assertions and C is a command. P is called precondition and Q postcondition. It is possible to describe the change of a computer program with such a Hoare triple. Given the precondition P, when command C is executed, then the postcondition Q should hold. Furthermore, the Hoare logic introduces axioms and inference rules for proofing (partial) correctness. However, these constructs are not used in common BDD frameworks like Cucumber [10] or Behave [11].

Developers state tests in the form of Hoare triples. The basic skeleton of a test is given by the three key words Given (precondition P), When (command C) and Then (postcondition Q). In the following, we list an example of a behavior – here termed scenario – of a coffee maker. If customers choose to add sugar to their coffee, then the coffee maker should add sugar to the coffee.

```
Scenario: Coffee maker can add sugar to coffee
Given customer chose sugar
When customer presses OK button
Then coffee maker adds sugar to coffee
```

IV. BDD IN MALWARE ANALYSIS

This section describes in detail how BDD can be applied to malware analysis. By applying BDD to malware analysis, we overcome the problems that analysts face when reimplementing malicious behavior (cf. section II).

A. Overview of the Process

In this section, we detail how we enhance malware analysis with a testing-driven process – Behavior-Driven Development – in order to overcome limitations such as unreadable or undocumented code. From a reverse engineering point of view, our approach is mainly a top-down approach [12]. Our goal is to dive more and more into the details until the behavior is reimplemented. Behavior that is yet not known is mimiced with mock objects [13]. The process consists of an initialization phase and a cycle of four steps that is traversed repeatedly. It is similar to the phases of TDD or BDD, however analysts have to derived the specification from the malware sample by themselves.

During the initialization phase malware analysts set up their analysis and development environments. Then they pinpoint the behavior of interest in the binary, i.e. identifying the interfaces that mark the entry point and exit points of this behavior. Once they have set up everything, they enter the reimplementation phase. Figure 1 gives an overview of the reimplementation phase. It consists of four individual steps that are traversed repeatedly. The objective of each pass is reimplementing one submodule of the behavior of interest at a time. At first this submodule is observed and its functionality is analyzed by using the top-down approach. In the second step, malware analysts reflect on their observations and express them consistently by writing down a test case in natural language. Thirdly, they write just enough code in order to pass this test. For it, they analyze in detail the relevant sections of the binary. In the fourth step, the code that has just been written for passing the test is refactored. For example, coding style or the design are improved. At the end of the fourth step, the cycle can be left in case the initial end-to-end acceptance test passes. In the following sections, we detail each phase.

B. Pinpointing the Behavior and Initial Acceptance Test

In this preliminary phase, analysts setup their analysis and development environment. Then they proceed to search for the behavior in the binary. There must exist a region in the binary where this behavior starts, in the following called entry point $\mathbb{S}$. Also there must exist one or several regions in the binary that mark the successful or unsuccessful termination of this behavior. These are termed exit points in the following and they are denoted by $\{\mathbb{E}_0, ..., \mathbb{E}_n\}$, where $n \in \mathbb{N}$. Once $\mathbb{S}$ and $\{\mathbb{E}_0, ..., \mathbb{E}_n\}$ are known, an initial end-to-end acceptance test should be written. This acceptance test passes as soon as the behavior has been reimplemented. It serves therefore as a guide during the implementation phase. For this initial acceptance test, malware analysts have to capture test data. Since this initial acceptance test should test the behavior as a black box, they have to capture test data at $\mathbb{S}$ and $\{\mathbb{E}_0, ..., \mathbb{E}_n\}$.

For illustration purposes, we assume a decryption routine of a symmetric cryptographic algorithm. This decryption routine has one entry point $\mathbb{S}$ and one exit point $\mathbb{E}_0$. This routine takes as input a pointer to an encrypted buffer, the size of the encrypted buffer and outputs a pointer to a buffer that contains the decrypted data. In order to capture test data, a malware analyst would set two breakpoints – one at $\mathbb{S}$ and one at $\mathbb{E}_0$ – and let the sample run. At each breakpoint, they would extract the corresponding data. They would use this data then in their initial acceptance test. Once they have reimplemented the decryption routine, the reimplementation should be able to successfully decrypt the encrypted buffer.

1) Example Pinpointing DGAs: Domain generation algorithms (DGAs) are a fallback mechanism in case the original command and control server has been taken down. In this

case, the bot generates a set of domains that it tries to contact. Today, the majority of DGAs is time-based. Hence, the bot has to get the current time for generating domains. For example, Windows offers information about the current time via the API call *GetSystemTime*. Setting a breakpoint on this function and following the data flow of the time information might lead to the beginning of the DGA.

2) Example Pinpointing Command Dispatchers: Per definition, a bot executes commands sent/received by/from its botmaster [14]. Bots implement typically more than one command, often times more than ten commands. A command dispatcher executes the commands issued by the botmaster. Since the behavior of a bot is defined by the commands it can execute, finding the command dispatcher and understanding its individual commands leads to the understanding of the capabilities of the bot in general.

An easy way of finding a command dispatcher is to follow the data flow of incoming messages. We assume that communication is done via the network. Therefore, a reasonable starting point is observing system calls/API calls that are related to receiving messages from the network. From there analysts have to follow the data flow of the received data. Common bots decrypt and verify the network data, parse the decrypted message and finally execute the received commands. The decision is taken in the command dispatcher. A telltale sign of a command dispatcher is the usage of a switch statement.

C. Step 1: Observing the Behavior

Once we have set up the initial end-to-end acceptance test, we can dive into the details of the behavior of interest. Now we are entering a cycle that consists of four steps. The first step is observing a part – in the following termed submodule – of the behavior of interest. This can be, for example, the key setup of a cryptographic algorithm. At first, we are interested in getting a rough overview and also in determining a submodule's entry and exit points. Therefore, it is important to use a Top-Down approach here. A Top-Down approach aims at forming the big picture of a system. This is achieved by repeatedly breaking a system down into smaller subsystems until it cannot broken down further. At first subsystems are treated like a black box. In later steps these black boxes are opened and broken down into further subsystems.

D. Step 2: Writing a Test

Once the submodule has been observed, malware analysts write a test for this submodule. They describe in this test description what they have observed. The main goal is to concisely describe what this submodule does. Furthermore, a corresponding Hoare triple in Given-When-Then-form has to be stated.

A fundamental part of our approach are mock objects [13]. Mock objects mimic the behavior of real objects such that they interact with other code in a controlled way. In software development, mock objects are among other use cases used for replacing non-existing objects. In our case, these non-existing objects are submodules that are yet not 100% understood. Therefore, we simulate these objects with mocks.

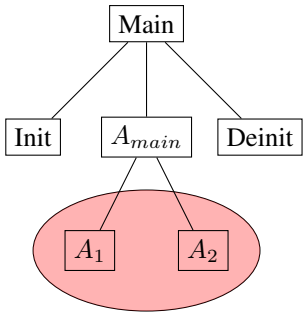


Fig. 2: Illustration of the Top-Down approach and the usage of mock objects. When reimplementing submodule A, the main logic A_{main} should be reimplemented first. A_1 and A_2 should be replaced by mock objects in the meanwhile.

Figure 2 shows the call graph of a simple behavior. The main logic of the behavior is executed in the main function. Main calls an init function, another submodule A and a deinitialization function. A_{main} represents the main logic of submodule A. From there two other functions (A_1 and A_2) are called. Assume that the next step would be implementing submodule A. Since we are using a Top-Down approach, we reimplement at first the main logic A_{main} of submodule A. However, A_{main} depends on A_1 and A_2. This means that these two functions have to be replaced by mock objects. We gather test data at their interfaces and let corresponding mock objects return this test data.

E. Step 3: Making the Test Pass

Next, malware analysts write just enough code for making the test pass. In this step, the malware analysts consult the malicious binary. Since this binary serves as valid system specification. They can follow the control flow in a debugger in detail or consult their disassembler and follow multiple control flows statically. It is import to stay focused during this step and to just reimplement the code that is needed for passing the test. Optimizations of the code should not be part of this step, this can be done in the next step. The main focus is on making the test pass. Here, the old rule *Premature optimization is the root of all evil* should be followed.

F. Step 4: Refactoring the Code

The fourth step ensures the conciseness and readability of the code. Therefore, analysts refactor their code. Refactoring is the process of altering the syntax of a software system without altering its semantics in order to improve its internal structure [15]. As in the original BDD, it is a mandatory step. Since there are already tests, bugs that are introduced during this step are typically found very quickly. This gives analysts the confidence to rigorously improve their code. Refactoring is a very broad field. Fowler describes basic refactorings in [15]. The goal is describing computations in a (more) human readable way. Therefore, analysts have to reverse the optimizations that were introduced by compilers or the obfuscations that were introduced by obfuscators. These optimizations include refactoring inlined code (e.g. memcpy) or inlined loops, represent complex expression with the help of intermediate results and removing dead expressions.

If after this step the initial end-to-end acceptance test passes, then the behavior of interest has been successfully reimplemented. If this test still fails, then the analyst has to do another round of this four step cycle in order to advance towards the goal.

V. CASE STUDY: NYMAIM

In this case study, we reimplement the DGA of Nymaim by applying BDD to malware analysis. Nymaim is mainly a malware dropper with further capabilities such as stealing passwords or a SOCKS proxy. Originally, it emerged in late 2012. Additionally to packing, Nymaim is highly obfuscated [16]. Nymaim's DGA is time-dependent, i.e. each day the results are different. However, it is deterministic and does not change in different environments.

A. Environment Setup

We used two Windows VMs as analysis environment. The first VM was a Windows XP SP3 x86 machine. This VM was just used during blackboxing. The second VM was a Windows 7 SP1 x86 machine. We used IDA Pro 6.8 as disassembler [17], Immunity 1.85 as debugger [18] and Mandiant ApateDNS 1.0 for spoofing DNS responses [19]. We reimplemented the DGA in Python 2.7 [20]. For writing tests, we used the BDD framework behave [11].

B. Pinpointing the Behavior

We let the sample run in the Windows 7 virtual machine in order to observe its initial behavior. At first, we noticed that it resolved *google.com*. We assumed this to be a typical connectivity test. Then, the sample resolved some hardcoded domains. Since the VM was not connected to the Internet, the sample could not phone home. Immediately afterwards, we observed how the sample resolved generated domain names (e.g. *yjcmub.info*). We reset the VM and changed the date. We could observe the same behavior. However, the generated domains were different. We assumed that the underlying DGA is most probably time-dependent. Executing the sample in the Windows XP VM with a different system configuration but with the same date yielded to the same results. Therefore, we assumed that the algorithm is most likely deterministic.

Given these assumptions, we stated the hypothesis that the seed for this algorithm must come from a time source like the Windows API call *GetSystemTime* or the x86 instruction *RDTSC*. First, we set a breakpoint on the API call *GetSystemTime* and let the sample run again. This breakpoint was hit after the sample contacted its hardcoded domains. From there we followed the control flow and we were able to pinpoint algorithm's exit $\mathbb{E}_0$, where 30 generated domains are outputted. Hence, the input is the SYSTEMTIME structure returned by *GetSystemTime* and the output is a list of 30 generated domains.

We captured in the debugger a valid input/output pair at the beginning/end of the behavior. Then we used this input/output pair for a first end-to-end acceptance test. This acceptance test failed of course at this point in time. But throughout the reimplementation phase it served as an indicator of task completion.

```
Feature: Generate domains time-dependently
In order to have a fallback machanism in case
the hardcoded domains have been taken down,
Nymaim should be able to generate domains for
establishing a command and control chanel

Scenario: Nymaim DGA computes domains
Given the day is "2015-06-12"
When DGA computes domains for this date
Then the domais for this date are
| domains       |
| dmjdfotcy.in  |
| yjcmub.info   |
| uiismpexr.info |
| rszsgpzivi.info |
| pratyequtgd.ru |
[...]
```

C. Entering the Cycle

We had already determined the entry and exit points of the behavior. In addition, we had an end-to-end acceptance test for verifying that the behavior is implemented according to the malware sample's system specification. Now it was time to dive into the details. While stepping through the code in a debugger, we could see that there is an initialization of the DGA and that there are two other algorithms. The first algorithm is a pseudo-random number generator (PRNG). The second algorithm constructs the domains. It does so by drawing the length L of the domain, then drawing L characters from the alphabet $\{a, b, c, ..., z\}$ and finally adding one of five top-level domains (TLDs). An example of the outcome would be *rszsgpzivi.info*. This is actually the main logic of the DGA.

Since we are using a Top-Down approach, the first component to be reimplemented is the main logic. As described above, the main logic consists of three steps. Each step is targeted individually. Explanatory, we reimplemented the step where the TLD is chosen. Observing this step yielded to a basic understanding: this step draws a number from the PRNG and uses a switch statement for choosing a TLD. We came up with the following test case:

```
Scenario: Nymaim DGA chooses correct
TLD from set of possible TLDs
Given the seeds
| seed     |
| 78670654 |
| 44370352 |
| 35461477 |
| 97912344 |
When DGA computes TLD
Then the TLD is ru
```

This yielded to the following python code where the PRNG and SEEDS are still mock objects.

```
def computeTld(self):
  modulo = 600
  tld = ["ru","net","in","com","xyz", "info"]
  eax = PRNG(SEEDS, modulo)
  return tld[eax]
```

The other two steps of the main logic can be implemented respectively. Next it was time to focus on the PRNG. So far we had replaced this component with mock objects. We entered the cycle once more. While observing this component,

we noticed that it takes as input an integer number that is used as modulo and four seeds. It outputs an integer in the closed interval $[0, (modulo - 1)]$. Care had to be taken since the PRNG has side-effects on the inputted seeds. The following listing shows a test for the PRNG.

```
Scenario: PRNG works correctly
for given seeds and modulo
Given the modulo 600
And the seeds
| seed       |
| 123172080  |
| 79962903   |
| 133504895  |
| 2326822159 |
When PRNG executes
Then the output is 1
```

Once the test had been implemented, we dived into the details of this component and wrote code to pass it. The following code passes the test.

```
def execute(self, seeds, modulo):
  a = seeds[0] << 11 ^ seeds[0]
  b = seeds[3] >> 19 ^ seeds[3]
  a = b ^ a ^ (a >> 8)
  c = seeds[2]
  self._updateSeeds(seeds, a)
  return (a + c) % modulo / 100
```

Please not that it would be very interesting to have some kind of smoke test for such a mathematical function. For example, a naive approach for generating a lot of test data would be using Ida Pro's Appcall (cf. Future Work in section VIII).

Since the initial end-to-end acceptance test did not pass after this iteration, another round of the cycle had to be run through. The next step would have been implementing the seed update. However, due to space restrictions, we will skip the next iterations.

D. Using the Extracted Behavior

Manual searches on the Internet revealed that Nymaim's DGA results in a lot of collisions. Because samples from different years resolved the same domains. We used therefore the reimplemented DGA in order to estimate how many collisions this algorithm actually yields to within a ten year period. Figure 3 shows the total domains generated vs. the total colliding domains generated by Nymaim's DGA. A colliding domain is a domain that is not unique during the considered period. This figure shows that the DGA generates around 120.000 domains after 10 years. However, around 100.000 domains are not unique within this ten year period. Interestingly, collisions occur only after roughly six months. After six months the collisions steadily increase. The collisions start only after six months due to the modulos used by the DGA. Unfortunately, we cannot state whether the developers of Nymaim aimed at having a lot of collisions in their DGA or this is just a bug.

E. Discussion

In this case study, we have guided the reader through the whole process of BDD applied to the malware analysis process. The outcome is code that is developed according to the malware sample's system specification, concise, readable and well documented. We have written one end-to-end acceptance test and five additional scenarios that ensure the correctness of the DGA's subfunctionality. The implementation of one piece of the DGA at a time and replacing not yet existing pieces by mock objects enables malware analysts to focus only on relevant code. This keeps the window of code to be analyzed at a time small and also supports the understanding of the code.

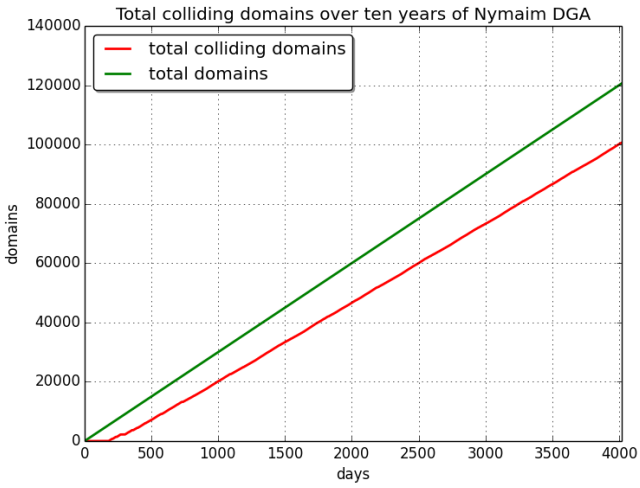


Fig. 3: Nymaim's DGA total domains generated vs. total colliding domains within a time period of ten years

VI. LIMITATIONS

We discuss limitations of BDD in malware analysis in this section. A major objection against our approach might be the decrease in time efficiency. For example, malware analysts have to write testing code, they have to capture test data or they have to clean up the production code. It might be faster to just write hacky code. However, we believe that this extra time invested pays off. Our approach eliminates the problems described in Section II. So far there are no estimates of at how much overhead this process comes with. Malware analysts with experience in this process can minimize the overhead to a negligible amount of time. We consider the estimation of the efficiency of our approach as future work. We would like to point out that TDD faces similar objections. In industrial case studies at a major software company, it was estimated that the overhead of TDD ranges from 15% to 35% [1]. However, the tests served as documentation of the code and a significant increase in code quality was also measured. Another objection might be that our approach draws its main idea – using a *-driven development process – from software development. One might argument that a major requirement in this field is reusability. Code that malware analysts write does not have to be clean, tested and well documented since it will be discarded quickly. On the one hand, there are many long-running projects – with many contributers – that incorporate a lot of code written by malware analysts. In this case, you want to have code that is tested, readable and documented. On the other hand, these are not the only advantages of our approach. In

addition, our approach helps malware analysts to increase their focus. For example, by fading out non-relevant binary code with the help of mock objects, malware analysts focus only on the relevant parts. This might also lead to a speed up and counteract the above mentioned decrease in time efficiency due to, for example, writing tests.

VII. Related Work

In this section, we discuss related work in the fields of software testing and extraction of malicious behaviors.

A. Software Testing

The need for testing software emerged early. Since 1957 requirements of software systems have been tested [21]. Software testing was also an integral part of the waterfall model that was first mentioned in 1976 [6]. Test-Driven Development emerged from extreme programming in 1999 [7]. Several industrial case studies on the benefits of TDD exist (e.g. [2] or [1]). Dan North proposed Behavior-Driven Development in 2006 [8]. Van Lindberg describes how test-driven development can be used in open source development teams that write replacements for proprietary software [22]. In his approach, two teams work together. The first team is the specification team that provides test suites and the development team that implements code for these test suits. This method ensures that copyrightable expressions are screened from the development team. Durelli et al. propose a process for reengineering legacy systems that is aided by TDD [23]. However, their approach assumes that the source code is available in a higher-level language and also the documentation can be consulted.

B. Extraction of Malicious Behaviors

Caballero et al. presented an approach for binary code extraction [4]. Their approach identifies assembly functions interfaces and extracts the corresponding code. Please note that behaviors of interest are not limited to assembly functions and that obfuscation schemes remove functions or introduce esoteric calling conventions. Kolbitsch et al. proposed Inspector Gadget [3]. An automatic system for extracting algorithm of certain malicious behaviors from binaries. These algorithms are isolated, extracted and embedded in a stand-alone executable. They proof their approach to be feasible in several case studies. However, their approach has various limitations that make it impracticable. The authors of [24] presented a similar approach to Inspector Gadget that can extract domain generation algorithms from malicious binaries. Even though they show the feasibility of their approach, it suffers from similar shortcomings like Inspector Gadget.

VIII. Future Work and Conclusion

In this paper, we have shown how Behavior-Driven Development can be integrated into the malware analysis process. As in classical software development, it enhances the malware analysis process with readable, documented and tested code that has been developed closely to the behavior of interest's system specification given by the malware sample.

Future work will focus on user studies for identifying improvements and common problems. Furthermore, several tools that could make the process faster will be implemented. Such tools include a tool for acquiring test data and a smoke test generator.

Appendix A

Let us assume that some binary code was compiled with compiler C and assembled by assembler A. When this binary code is disassembled with disassembler D and decompiled with a decompiler B then this will result in at least syntactically different code. In the worst case, something went wrong in the decompilation and the result is semantically different. For illustration purposes, let have a look at Google Translate [25]. While the quality of its output is normally acceptable, the result of a translation chain from language L_1 over L_2 and L_3 back to L_1 produces at least syntactically different translations. For example, the German phrase "Der Tag, an dem die Mauer fiel, war historisch." is translated to Dutch as "De dag dat de Muur viel, was historisch." and from Dutch to Afrikaans as "Die dag dat die Muur geval het, was histories.". When translated back from Afrikaans to German it yields to "An diesem Tag, dass die Wand Fall war Geschichte.". This phrase is not only syntactically different but also semantically. Actually, it makes no sense at all. Please note that all three languages belong to the same language family. Similar effects can be observed, for example, when compiling C code with Visual Studio and decompiling this code to Pascal.

Acknowledgment

We would like to thank all our colleagues for the fruitful discussions. Also, we would like to thank Robert Cecil "Uncle Bob" Martin for spreading the word.

References

[1] T. Bhat and N. Nagappan. Evaluating the efficacy of test-driven development: Industrial case studies. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, 2006.

[2] E.M. Maximilien and L. Williams. Assessing test-driven development at ibm. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, 2003.

[3] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *Security and Privacy (SP), 2010 IEEE Symposium on*, 2010.

[4] J. Caballero, N. M. Johnson, S. McCamant, and D. Song. Binary code extraction and interface identification for security applications. In *NDSS*, 2010.

[5] R.G. Hamlet. Testing programs with the aid of a compiler. *Software Engineering, IEEE Transactions on*, 1977.

[6] T. E. Bell and T. A. Thayer. Software requirements: Are they really a problem? In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, 1976.

[7] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[8] D. North, Introducing BDD. http://dannorth.net/introducing-bdd. visited on March 18, 2016.

[9] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[10] cucumber - BDD framework. https://cucumber.io. visited on March 18, 2016.

[11] behave - BDD framework. http://pythonhosted.org/behave. visited on March 18, 2016.

[12] H. A. Mller, M. A. Orgun, S. R. Tilley, and J. S. Uhl. A reverse engineering approach to subsystem structure identification, 1993.

[13] T. Mackinnon, S. Freeman, and P. Craig. Endo-testing: unit testing with mock objects. In *Extreme programming examined*, 2000.

[14] D. Plohmann, E. Gerhards-Padilla, and F. Leder. Botnets: Detection, measurement, disinfection & defence. *The European Network and Information Security Agency (ENISA)*, 2011.

[15] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[16] D. Plohmann. Patchwork: Stitching against malware families with ida pro. In *Spring 9*, 2014.

[17] Hex-Rays. IDA Pro. https://www.hex-rays.com/products/ida/, 2015 Last access: March 18, 2016.

[18] Immunity Inc. Immunity debugger. http://www.immunityinc.com/products/debugger/, 2015.

[19] Fireeye. Mandiant ApateDNS. https://www.fireeye.com/services/freeware/mandiant-apatedns.html, 2015 Last access: March 18, 2016.

[20] G. Van Rossum and F. L. Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[21] D. Gelperin and B. Hetzel. The growth of software testing. *Commun. ACM*, 1988.

[22] Van Lindberg. *Intellectual Property and Open Source: A Practical Guide to Protecting Code*. O'Reilly Media, 2008.

[23] S. De Sousa Borges V. Durelli, R. Penteado and M. Viana. An iterative reengineering process applying test-driven development and reverse engineering patterns. In *INFOCOMP*, 2010.

[24] T. Barabosch, A. Wichmann, F. Leder, and E. Gerhards-Padilla. Automatic extraction of domain name generation algorithms from current malware. In *STO-MP-IST-111*, 2012.

[25] Google. Google translate. https://translate.google.com, 2015 Last access: March 18, 2016.