CACEE: CONTEXT AWARE CONCOLIC EXECUTION ENGINE FOR MALWARE ANALYSIS

A Thesis
Presented to
The Academic Faculty

By

Samuel Lovejoy

In Partial Fulfillment
of the Requirements for the Degree
Masters of Science in Electrical and Computer Engineering
School of Electrical and Computer Engineering

Georgia Institute of Technology

May 2021

CACEE: CONTEXT AWARE CONCOLIC EXECUTION ENGINE FOR MALWARE ANALYSIS

Thesis committee:

Prof. Brendan Saltaformaggio, Advisor
School of Computer Engineering
Georgia Institute of Technology

Prof. Frank Li
School of Computer Engineering
Georgia Institute of Technology

Mr. Chris M Roberts
CIPHER Lab
Georgia Tech Research Institute

Date approved: April 30, 2021

ACKNOWLEDGMENTS

I would first like to thank Prof. Brendan Saltaformaggio who heads the Cyber Forensics Innovation Laboratory at Georgia Tech. Without his help, I am quite confident that I would be stuck in a ”dead field” and have never discovered a passion for computer security. In addition, much thanks is owed to Mingxuan Yao, Ranjita Pai Kasturi, and Jon Fuller, the infallible charismatic trio of PhD students who kept my spirits up through weeks or even months of slow progress.

Next, much credit is owed to my family, who not only let me move back home in light of the coronavirus pandemic but also supported my education at every turn. Rachel Parent, who took care of me for such a long time and whom I call far too little. My friends, both here in Atlanta and back in Maine, you all are fantastic and while our paths are sure to split ways, I do wish you the absolute best.

Lastly, credit is due to Sandia National Laboratory, whose funding has sponsored this research. Years of work and experience are reflected in this document; it feels as though an entire four year college experience has been summed up in less than fifty pages. I am grateful for such a wonderful college experience at Georgia Tech, and doubly grateful for all the extraordinarily talented people I have met while here. See you all soon again.

TABLE OF CONTENTS

Acknowledgments iii

List of Tables viii

List of Figures ix

Summary x

Chapter 1: Introduction 1

Chapter 2: Background 3

2.1 Static Analysis 3

2.1.1 Overview 3

2.1.2 Control flow 3

2.1.3 Static Analysis Deterrents 4

2.2 Concrete Analysis 5

2.2.1 Overview 5

2.2.2 Data Dependence 5

2.2.3 Concrete Analysis Deterrents 5

2.3 Concolic Analysis 7

2.3.1 Overview 7

2.3.2 Function Modeling 7
2.4 Web APIs 8
2.4.1 Web API Components 8
2.4.2 API Keys 9

Chapter 3: Design 10

3.1 Dataset 10
3.1.1 Dataset 10
3.1.2 Dataset Restrictions 10
3.2 Code Coverage 11
3.2.1 Targeting 11
3.2.2 Greedy Exploration 11
3.2.3 Branch Prediction 12
3.2.4 Rewinding 13
3.3 Data Modeling 13
3.3.1 Instruction Modeling 13
3.3.2 Function Modeling 14
3.4 Web Profiling 14
3.4.1 Authentication Mechanisms 15
3.4.2 Packet Reconstruction 15
3.5 Rules 16
3.5.1 Library Call Context 16
3.5.2 Rule structure 17

3.5.3 Rule methods 17
3.5.4 Redundant Functions 18

Chapter 4: Implementation 19
4.1 Environment 19
4.1.1 Analysis Environment 19
4.1.2 Repositories 19
4.2 Choice of Tooling 19
4.2.1 QEMU & KVM 19
4.2.2 PIN 20
4.2.3 Triton 21
4.2.4 Python 22
4.2.5 Z3 22
4.2.6 Other Tools 22
4.3 Expanding the Triton Dynamic Analysis Framework 22
4.3.1 Exposing Additional Bindings to Python 23
4.3.2 Adding Support for Symbolic Pointers 23
4.3.3 Adding Support for Loaded Symbols 24
4.4 Creating Dataset Rules 24
4.4.1 Service Rule 25
4.4.2 API Key Rule 26
4.4.3 Downloader Rule 28
4.4.4 Dropper Rule 28

4.4.5 Exfiltrator Rule 29

Chapter 5: Evaluation 31

5.1 Vidar Case Study 31

5.1.1 Vidar Summary 31

5.1.2 Web Profile for IP API 32

5.1.3 Analysis on Vidar 32

5.2 LOWBALL Case Study 34

5.2.1 LOWBALL Summary 34

5.2.2 Web Profile for Dropbox 35

5.2.3 Analysis on LOWBALL 35

5.3 Vflooder Case Study 36

5.3.1 Vflooder Summary 36

5.3.2 Web Profile for VirusTotal 37

5.3.3 Web Profile for Twitter 37

5.3.4 Analysis of Vflooder 37

5.4 Findings 38

Chapter 6: Conclusion 39

6.1 Related Works 39

6.2 Limitations 39

6.3 Conclusion 40

References 42

LIST OF TABLES

2.1 A non-exhaustive list of static analysis deterrents. 4

2.2 A non-exhaustive list of concrete analysis deterrents. 6

3.1 Unhandled corner cases for the system design. 11

3.2 The function model for the function WinHttpReadData. 14

3.3 The four authentication and authorization schema. 15

3.4 The elements that make up library call node. 16

4.1 Rules for modeling behaviors within the cloud-abusing malware dataset. . . 24

5.1 CACEE analysis for web rules on three case studies. 38

5.2 CACEE analysis for behavior rules on three case studies. 38

LIST OF FIGURES

2.1 An example of a basic block with its respective defuse chain. 6

2.2 A sample API request to the OMDB API with JSON response. 8

3.1 An artificial control flow graph demonstrating greedy exploration. 12

3.2 A sequence of three API calls that indicate downloading and executing a file. 17

4.1 The code for a HTTP GET request using WinINet and the resulting packet. . 27

5.1 Decompiled code from the Vidar malware simplified for readability. 33

SUMMARY

An emerging pattern in malware is the use of public web services for command and control (C&C) infrastructure. This new trend, combined with the short lifespan of malware in the wild, makes extracting behaviors from malware in an automated fashion a difficult problem. The Context-Aware Concolic Execution Engine (CACEE) is a tool designed to recreate the original execution context, forcing Windows 32-bit malware to execute their payloads as if they were still operational. CACEE monitors the flow of data as the payload executes, and uses this information to synthesize the behaviors the malware exhibits. Three malware case studies that abuse public web services are analyzed with CACEE, and the results are compared against manual reverse engineering.

CHAPTER 1
INTRODUCTION

The lifespan of malware in the wild is typically very short, normally lasting only a few days before an updated version is released or activity stops altogether [1]. After this narrow timing window has expired, it is difficult for automated tools to analyze the payload of the malware. Trigger conditions, such as temporarily registered domain names, stop functioning. As a result the malware does not execute its payload and now security analysts must manually reverse engineer the malware in order to ascertain any malicious behaviors.

This already tedious process is often made worse by the fact that malware authors incorporate numerous tricks in order to deter security researchers from investigating their malware. The tricks, collectively called anti-reverse engineering (anti-RE) techniques, also prevent a wide array of existing tools from functioning properly. Security researchers will update their tools or develop new tools to combat anti-RE techniques. Malware authors then respond by further obfuscating their malware, inventing new tricks and trends to halt the efforts of security researchers. New tooling must be able to handle current trends in anti-RE techniques in order to generalize well.

One staple of malware is the use command and control (C&C) infrastructure [2]. C&C servers are used by malware authors to control infected machines. They are most often tied to domains, and as a result a staple of cyber defense has been to use the plaintext domain name service (DNS) to analyze network traffic and blacklist domains if they are flagged as malicious. Historically, malware authors may employ a domain generation algorithm, which procedurally generates new domains and sends them out to infected hosts as old ones are blacklisted [3, 4]. A new trend in malware is to use public web services as C&C infrastructure.

Public web services, such as Dropbox and Google Drive, offer enormous utility to the

general public, including malware authors. Because of their reputation and popularity, network administrators cannot blacklist these highly trafficked websites. While most cloud services implement automatic scanning for malware samples, malicious actors implement many workarounds for this, some as simple as just uploading shell commands or partitioning files [5]. Malware abuse public web services for network discovery, anti-RE techniques, botnet management, and even reverse shells [6].

An analysis of web service abusing malware could provide insights on how to create the next generation of cyber defense. Existing network-oriented systems such as intrusion detection systems rely on network signatures to proactively secure networks [7], which can be circumvented by utilizing public web services. With a large-scale analysis of web service abusing malware, new signatures could be generated for host-oriented systems. In addition, service providers can potentially take action against malicious API keys or accounts associated with malicious activity.

With such a short time window to collect very elusive samples, a novel system is proposed to recreate the context that the malware originally used for its payloads when it was first encountered in the wild. This system is to be driven by concolic execution. Concolic execution is a technique that involves executing the malware and exploring now unreachable code sections— analyzing malware as if it were a code coverage problem.

To this extent, we propose the Context-Aware Concolic Execution Engine (CACEE). A tool designed to solve past the trigger conditions present in public web service abusing malware. Beyond this, CACEE aims to analyze each payload of a given web abusing malware, and tag samples with abstract behaviors in an automated fashion. CACEE can then categorize samples based on behaviors on datasets consisting of no longer operational malware.

CHAPTER 2

BACKGROUND

2.1 Static Analysis

2.1.1 Overview

Static Analysis is the practice of analyzing the bytes of an executable. In practice, this is done by analyzing the disassembly. Most modern Windows PE32 malware are compiled through the Microsoft Visual C++ Compiler (MSVC) or through Microsoft .NET [1]. One of the most important parts of static analysis is symbol identification. There are two types of symbols used by binaries— dynamic symbols and debugging symbols. The linker uses dynamic symbols to load in library functions into memory then store their pointers in the global offset table. Debugging symbols, on the other hand, are optional symbols used to convey excess information when debugging a program. In practice, only dynamic symbols are used for malware analysis since the malware author will omit debugging symbols. In the context of malware analysis, static analysis is very fast but lacks the accuracy compared to concrete analysis.

2.1.2 Control flow

One of the main incentives for using static analysis is to ascertain control flow. In the x86 architecture, there are four assembly instructions that can impart changes to control flow, since the instruction pointer is not directly writable [8]. These four instructions are jumps, calls, repeats, and returns. When constructing a control flow graph— a jump indicates a potential fork in control flow— a branch. The destination of a call instruction marks the start of a function. On the other hand, a return instruction marks the end of a function (note that a function may only have one start but may multiple ways to exit). Repeat instructions

Table 2.1: A non-exhaustive list of static analysis deterrents.

Technique	Summary
File Compression	The executable decompresses itself as it runs.
Runtime Encryption	The malware decrypts its own code as it runs.
Dynamic Memory	The malware tracks most or all of its data on the heap.
Missing Code	The payload is hidden in other locations (files, internet, etc).
Static linking	Statically linked library code removes symbol information.
Segment Manipulation	Section header properties are manipulated at runtime.
String Obfuscation	String constants are decoded during execution.
Self-Modifying Code	The malware modifies its own code during execution.
Garbage Data	The malware fills itself with red herring bytes.

do alter control flow, but will eventually proceed into the next instruction, so do not affect the control flow graph. Any set of sequential instructions with no breaks in control flow is called a basic block.

Dynamic symbols play an important role in control flow. Call instructions made to library code are sections in control flow where execution leaves the user memory space. This is often used to execute privileged instructions or to make privileged system calls [8, 9]. Since the parameters to these library calls have strict typing, security analysts can use library calls to determine data types. Moreover, since symbols represent code, they are also partial indicators of behavior.

2.1.3 Static Analysis Deterrents

There are numerous ways for a malware author to obfuscate their code from static analysis. Many existing code obfuscation tools implement one or more of these anti-reverse engineering (anti-RE) techniques; these tools are often collectively referred to as packers. A non-exhaustive table of static analysis deterrents is provided in Table 2.1.

2.2 Concrete Analysis

2.2.1 Overview

Concrete analysis (used interchangeably with dynamic analysis) is the practice of analyzing the behavior of an executable while it runs. Unlike static analysis, concrete analysis is able to monitor the flow of data throughout system memory and therefore provide high resolution data dependence. Data dependence involves tracking which registers and memory regions are used and defined by each instruction as they are executed. To do this, concrete analysis is often also accompanied by a debugger such as WinDbg [10, 11]. Like static analysis, there are numerous tricks a malicious program may employ in order to deter concrete analysis.

2.2.2 Data Dependence

By tracking register and memory states before and after instructions are executed, it is possible to accurately monitor the flow of data through a binary as it executes. This is called data dependence. For CISC architecture sets like x86, instructions must often be modeled individually as compared to RISC architectures like ARM where instructions can often be modeled in groups [12]. One of the most common representations of data dependence is a data structure called a define-use or def-use chain [13]. A def-use chain indicates which registers and memory regions are defined or used by an individual modeled instruction. An example of a def-use chain for a single basic block is provided in Figure 2.1.

2.2.3 Concrete Analysis Deterrents

Like static analysis, it is important to consider the many ways a malware author can implement countermeasures for concrete analysis. These techniques are tabulated in Table 2.2.

```
lea eax, [esp+0xe50]            load local variable into eax
push offset 0x404148            push pointer to string constant 'rb'
push eax                        push eax onto the stack
xor ebx, ebx                    set ebx = 0
call ds:fopen                   call fopen(local_variable, 'rb')
mov esi, eax                    move the result of fopen into esi
add esp, 8                      clean up the stack from fopen
test esi, esi                   check to see if fopen returned null
jz 0x401f23                     jump if fopen returned null
```

```
lea eax, [esp+0xe50]            D: EAX           U: ESP, [0x18fee8]
push offset 0x404148            D: ESP, 0x18f098 U: 0x404148
push eax                        D: ESP, 0x18f094 U: EAX
xor ebx, ebx                    D: EBX           U: EBX
call ds:fopen                   D: ESP, 0x18f090 U: 0x18f098, 0x18f094
mov esi, eax                    D: ESI           U: EAX
add esp, 8                      D: ESP           U:
test esi, esi                   D: EFLAGS        U: ESI
jz 0x401f23                     D:               U: EFLAGS
```

Figure 2.1: An example of a basic block with its respective defuse chain.

Table 2.2: A non-exhaustive list of concrete analysis deterrents.

Technique	Summary
Debugger Check	The malware checks to see if it is running under a debugger and adjusts behavior if so.
Artificial Exceptions	The malware introduces artificial exceptions and adjusts behavior if an attached debugger handles the exception.
Process forking	The malware spawns new processes and performs malicious operations in a new process.
Sandbox Check	By checking the hardware on its machine, the malware stops executing if it sees virtualized hardware.
Killswitches	A killswitch is in place that stops the malware from functioning once activated, often involving the internet.
Long Sleep Timers	The malware waits a very long time before running, or only activates on a specific day.
Exact Constraints	The payload is only executed given precise constraints such as timing or hardware constraints.
Hash-based Loading	The malware verifies every loaded image by hash, greatly reducing the effectiveness of function hooking.

2.3 Concolic Analysis

2.3.1 Overview

Concolic (concrete + symbolic) analysis is the practice of introducing constraints to manipulate system flags in order to adjust dynamic control flow throughout execution. Branch conditions are switched throughout execution so that user specified locations in the binary can be reached; every switched branch condition introduces one or more new states [14, 15]. The original purpose of symbolic execution was for code coverage, but use has since expanded to include malware analysis [16]. Without concrete analysis, symbolic analysis is highly prone to the path explosion problem. The number of states grows exponentially and analysis becomes very resource intensive [14]. Concrete analysis can be used to combine states and significantly reduce memory usage [17].

2.3.2 Function Modeling

Since a large percentage of the number of instructions executed are outside of the user space inside of library functions, modeling the behavior of these library functions can greatly reduce the amount of resources required for concolic execution [16]. In order to step over library functions, it is necessary to model which pieces of data the functions modify within the user memory space. This often requires manual modeling of functions, listing inputs and outputs and which data types are expected. The outputs to modeled functions often become constraints that the symbolic tool must symbolize to redirect execution.

In addition to modeling the behavior of library functions, dynamic binary instrumentation is also frequently necessary to enable concolic execution. Dynamic binary instrumentation involves inserting callbacks to specific instructions, functions, or binary images [18]. These callbacks are the primary means for implementing instruction modeling and function modeling.

```
curl --request GET \
 --url http://www.omdbapi.com/?apikey=[apikey]\&t=The+Matrix\&y=1999

{"Title":"The Matrix","Year":"1999","Rated":"R","Released":"31 Mar
1999","Runtime":"136 min", "Response":"True"}
```

Figure 2.2: A sample API request to the OMDB API with JSON response.

2.4 Web APIs

2.4.1 Web API Components

Many public web services offer APIs, both free and commercial, for developers and businesses. These APIs follow the client-server model, where the user sends an HTTP(s) request and the server then returns data for that request. The most common architecture for these APIs is the REST (representational state transfer) architecture, which emphasizes statelessness [19]. Some APIs require authentication, this is most commonly done by the OAUTH authentication framework [20].

While API requests vary from service to service, most requests contain four basic components: the HTTP verb, resource endpoint, additional HTTP headers, and form data. The HTTP verb is one of GET, HEAD, POST, PUT, DELETE, CONNECT, OPTIONS, TRACE, or PATCH [21]. While mostly important for the HTTP syntax, it can also differentiate endpoints. Second the resource endpoint, this is the name of the object on the server to request. Third are additional HTTP headers that are necessary to complete the request. These are typically outlined in the API documentation; API keys commonly appear here. Last is the form data and/or HTTP body, this contains data to be uploaded to the server, and typically only exists for HTTP POST actions. An example of API usage for the Open Movie Database is shown in Figure 2.2, showing a GET request and a JSON response [22].

2.4.2 API Keys

Many web APIs require authentication through user accounts. These accounts are typically represented in the form of an API key. API keys are a type of Uniform Resource Identifier (URI), which manifest themselves in code as strings [23]. While malpractice, API keys and other parameters like them are typically hardcoded by malware authors [24].

CHAPTER 3

DESIGN

3.1 Dataset

3.1.1 Dataset

When designing a tool, it is important to keep in mind the scope of the problem for which the tool is designed for. In this case, the scope of the dataset is MSVC C/C++ compiled portable executable 32-bit binaries (PE32). While some contemporary malware have begun using 64-bit, the majority of the existing samples are Windows PE32 [1]. Static analysis is used to select malware samples with interesting behaviors. In particular, the sample should contain dynamic symbols for internet functions. Additionally, the sample should have an existing VirusTotal report; this is used in order to establish the ground truth [25]. Most importantly the malware sample should connect to a popular domain as cited by Alexa's top visited sites [26]. While these are the only two strict requirements, samples are then prioritized based on file entropy and size.

3.1.2 Dataset Restrictions

In the making of an automated, general tool such as CACEE, some samples prove too sophisticated. As such, there are numerous problems with some binaries that require them to be discarded. These problems are tabulated in Table 3.1.

With these cases kept in mind, other anti-RE techniques shown in the previous chapter must be accounted for in the system design.

Table 3.1: Unhandled corner cases for the system design.

Problem	Rationale
Static Linking	Since static linking removes the majority of dynamic symbols within a binary. User code and library code cannot be easily distinguished complicating behavior analysis.
Process forking	While possible to follow new processes, it quickly becomes too cumbersome to handle reloading every tool in another memory space.
Self-Modifying Code	The program trace catalogues instructions by address, if two different instructions exist at the same address, then retracing becomes very difficult.
Segment Manipulation	Complex packing techniques such as segment manipulation often interfere with symbolic memory regions, since set permissions or symbolic memory regions is required.

3.2 Code Coverage

3.2.1 Targeting

Concolic execution is able to explore multiple paths beyond concrete execution. However, not all paths may be feasible at all and some may not be executable within a reasonable time frame. As such, it is important to consider which locations in the binary contain the payload or at least reach out to C&C infrastructure. An excess reliance on static analysis methods such as these may reduce the scalability of the system. With this in mind, CACEE only cross-references specific symbols found in these images: *wininet.dll, winhttp.dll, winsock.dll, winsock2.dll* [27]. Specific API calls of interest within these images include but are not limited to *InternetConnect*, *WinHttpReadData*, and *ConnectSocket*.

3.2.2 Greedy Exploration

CACEE relies on greedy exploration— a search algorithm where CACEE approaches basic blocks with stop conditions. Stop conditions are defined by program exit, unsolvable symbolic conditions, or if the malware fails to catch an exception. After a stop condition has been reached, CACEE rewinds the program state to the closest explorable symbolic

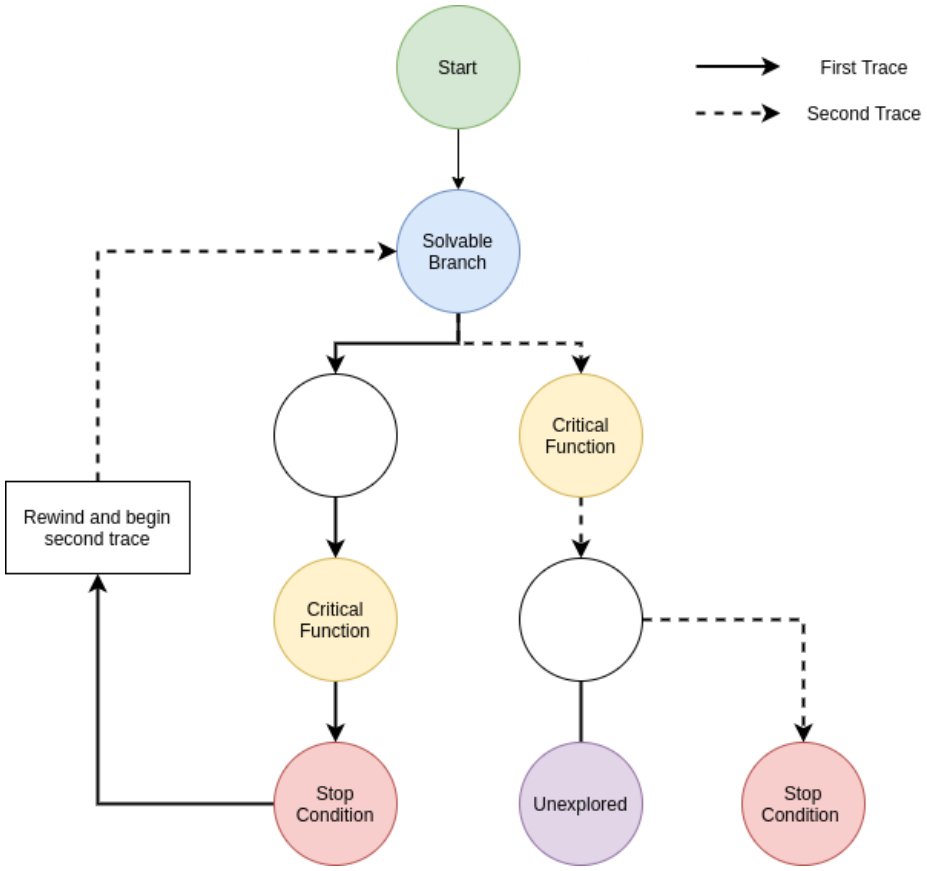

Figure 3.1: An artificial control flow graph demonstrating greedy exploration.

branch, then continues exploring on the new path. After all stop conditions or solvable branch constraints have been exhausted, exploration is considered complete. In this way, CACEE functions as a code coverage tool for functions of interest, rather than exploring the entire code segment. An illustration of greedy exploration is shown in Figure 3.1.

3.2.3 Branch Prediction

Since the introduction of symbolic data in greedy exploration leads to the path explosion problem, certain precautions must be put in place in order to limit the time spent solving for solvable branches. One solution to this is through branch prediction, a technique commonly employed into processors [28]. CACEE predicts the successive branch that leads to a better result based on the previous exploration of that branch [29]. In practice, this is often applied to non-critical validation functions, such as checksum functions. If a branch is encountered

more than once with different symbolic data, then CACEE remembers the previous branch conditions and uses these to predict a path out of the validation function, saving time and resources.

3.2.4 Rewinding

In order to reset execution to the last solvable branch, it is necessary to rewind the program state back. In order to do this, data dependence must be captured and stored for every instruction executed as well as any changes that the system makes to memory. Upon reaching a stop condition, data dependence from the program trace will be used to step backwards instruction by instruction. This is done in favor of using memory snapshots, since data dependence must be used anyways for behavioral analysis.

3.3 Data Modeling

Data dependence is a very important component of designing a concolic engine. To achieve high resolution data dependence, instructions and functions must be modeled so that the system understands the changes made to memory and registers. This is needed to populate the define-use chain— a data structure cataloging which registers and memory regions were defined or used for every instruction. In the case of instruction modeling, opcodes are stepped into, while in the case of function modeling opcodes are stepped over.

3.3.1 Instruction Modeling

Data dependence needs to be dynamically determined for each instruction. Since x86 is a complex instruction set, it is very cumbersome to model each instruction, yet each instruction must be modeled individually. CACEE wraps over existing tools for its instruction modeling, then implements its own callbacks whenever unsupported or unrecognized instructions are encountered.

Table 3.2: The function model for the function WinHttpReadData.

Function Name	`WinHttpReadData`
Parameter Data	`IN HINTERNET hRequest`
	`OUT LPVOID lpBuffer`
	`IN DWORD dwNumberOfBytesToRead,`
	`OUT LPDWORD lpdwNumberOfBytesRead`
Calling Convention	`stdcall`
Return Type	`boolean`

3.3.2 Function Modeling

Unlike instruction modeling, few robust tools exist that exhaustively model the behavior of the Win32 API. In order to step over library calls, CACEE requires a complete modeling of the Win32 API reference. This includes the function inputs and outputs, data types, enumerated values, and the contents of complex memory structures.

The solution CACEE utilizes is automatic and based at the source level. Automatic function modeling parses information from the Windows32 header files based on variable names from Microsoft's Source-code Annotation Language (SAL) [30]. The declarations of the typing in the SAL indicate whether the function parameters are inputs or outputs, as well as some data typing information for simpler data types. SAL provides CACEE with the function name, parameter names, parameter types, return types, calling convention, and whether the parameter is an input or output. Some typing information, namely for variably sized complex memory objects, remain inconclusive. Table 3.2 shows a function model for the function *WinHttpReadData*.

3.4 Web Profiling

In designing this system, one must consider how the malware author will implement their network behavior. For CACEE, network behavior is extracted through packet reconstruction from groups of API calls in the program trace. Additionally, the characteristics of

Table 3.3: The four authentication and authorization schema.

Authentication Method	Description
Optional Header	An API token is attached as an additional header of each HTTP(s) packet sent out.
Stateful API	An API token is sent to a dedicated endpoint and the server designates a session for the key.
Request Token	The malware author must request a new token in order to use the web API.
No Token	The malware uses only public resources from the web service.

the web API implementation affect the API calls the malware author will use. Any use of public web APIs by malware is considered malicious, regardless if the service is used in a benign fashion.

3.4.1 Authentication Mechanisms

There are numerous schema web APIs use to verify users; all web APIs analyzed in this document have dedicated endpoints for authentication, although authentication may be optional for public resources. Since the steps required for authentication and authorization vary between each web API, CACEE must handle each case [31, 32, 33, 25]. Within the context of CACEE, depending on the scheme, the malware author may opt to hardcode their API key. Different Windows C++ functions, such as *WinHttpAddRequestHeaders*, may be observed in one scheme that may not be present in other schema.

3.4.2 Packet Reconstruction

CACEE implements packet reconstruction by grouping API call sequences together based on shared handles. This is important to identify characteristics across many different types of web services. API calls are categorized as either source calls, sink calls, or additional calls, with a source call potentially having multiple sink calls. An example of a source call is *InternetOpen*, which creates the first handle to used by other functions in the API

Table 3.4: The elements that make up library call node.

Data Collected	Description
Name	The name of the function called.
Parameters	The parameters passed to the library function.
Defines	Memory regions used by the library function.
Uses	Memory regions used by the library function.
Index	The location in the trace where the function call occurs.
Return Value	The return value from the library call.
Address	The address in the binary where this function is called.

sequence. *HttpSendRequest* is an example of a sink call, as it signifies the end to a request. Any other modifications or intermediate handles to the API sequence such as *InternetConnect* or *HttpSetOption* are categorized as additional calls. Parameter fields, notably content types can then be recognized from the reconstructed packet.

3.5 Rules

CACEE uses a system of rules to identify the behaviors of the malware and the characteristics of the web service. The rules are split into behavior rules and web rules respectively for behaviors and characteristics. All rules in CACEE are determined through data collected from calls to Windows C++ library functions. CACEE groups these calls together and determines which rules a malware sample satisfies. Rules analysis occurs after the program has concluded execution and is configurable by the end user.

3.5.1 Library Call Context

CACEE gathers additional context whenever the malware calls a library function. The data dependence from a library call is very complicated, and is usually derived from manual and/or automatic function modeling. Using data dependence, library calls are grouped together into a graph with each call representing a single node in the graph. The edges of the graph are indicate data dependence.

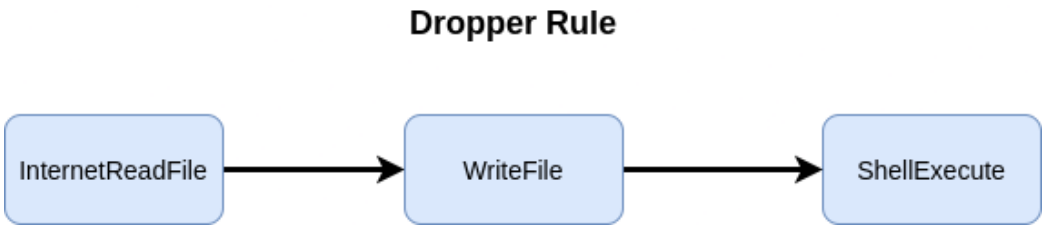

Figure 3.2: A sequence of three API calls that indicate downloading and executing a file.

3.5.2 Rule structure

Rules are built from two components, a structure and a method. The structure is derived from the *names*, *defines*, and *uses* of the library calls. Rule structures are small, general purpose graphs that represent potential Windows C++ API sequences that can indicate the behavior or characteristic. CACEE can then examine the program trace looking for groups of calls that could complete a rule structure. An example rule structure is provided in Figure 3.2. This particular structure of the dropper behavior indicates that the program downloads a file to disk then executes it. The arrows here represent data dependence (**not** call order), that is, data defined by *InternetReadFile* is used by *WriteFile* and so on.

3.5.3 Rule methods

Simply because there are three library call nodes linked by data dependence that satisfy a particular behavior does not necessarily indicate that the malware exhibits this behavior. In many cases there is more context needed; this context involves the *parameters* passed to each function and the *return value*. Additional methods are required to make sure that the matched behavior is truly exhibited by the malware. Consider the preceding example in Figure 3.2. It is possible that the program downloads a file to disk, but that file is not an executable file. Data dependence between *WriteFile* and *ShellExecute* may still exist; however, it is necessary to make sure that the path specified to *WriteFile* is the same as the path specified to *ShellExecute*. In this way rule methods greatly reduce the number of false positives that would appear should behaviors only be recognized as pure API sequences.

3.5.4 Redundant Functions

In many API sequences, there are precursor functions in order to open handles, establish permissions, etc. Within the behavior structure graphs, these functions are omitted since they do not include parameters of interest. The minimalist approach to building rule structure graphs fares better than including massive structures, where encoding schemes or functions from other libraries may overlap. An example of this in Figure 3.2 is *CreateFile*. This library call is necessary before a call to *WriteFile*, but does not contain any parameters of interest to the rule method.

CHAPTER 4

IMPLEMENTATION

4.1 Environment

4.1.1 Analysis Environment

The environment of choice is a Windows 7 Professional virtual machine. Security features on the virtual machine are removed, and the necessary tools are installed. QEMU and KVM are used alongside a prepared QCOW2 image to create the virtual machine [34]. Networking and USB components are virtualized. No compiled function hooking is performed. A fixed snapshot is used before every malware executes, such that changes to the registry and disk can be easily monitored.

4.1.2 Repositories

The samples are obtained through the public repository *VirusShare*, and have a report from *VirusTotal* available [25, 35].

4.2 Choice of Tooling

There are many ways in which one could implement a concolic code coverage tool. The tools we used best fit our needs and best fit our dataset, and may not necessarily be the most general purpose solutions. At large, it is possible to substitute one or more tools mentioned below.

4.2.1 QEMU & KVM

Since CACEE is a dynamic system for malware analysis, the malware must be run in a virtual machine. QEMU alongside KVM is the most advanced framework for virtualizing

Windows platforms on Linux. The configuration virtualizes a Windows 7 Professional 64-bit environment virtualizing disk, network components, keyboard and mouse peripherals, 8 gigabytes memory, and PCI bus for display.

4.2.2 PIN

The lowest level of CACEE is the dynamic binary instrumentation platform PIN [18]. PIN is a free, closed-source tool from Intel that runs in the same memory space as the target program. PIN just-in-time (JIT) compiles code for hooking, which is helpful against many dynamic anti-RE techniques. Furthermore, PIN is able to instrument instructions, functions, as well as images giving CACEE more flexibility. PIN has a few shortcomings, but all can be accounted for.

The first issue with PIN is that its API is in C/C++, which leads to longer developing times. This is mitigated by using Triton, which introduces PIN bindings to Python [36]. In this way, CACEE only needs to be recompiled when changes to the Triton source code are made. Some functionality must be added to Triton, however.

The second issue with PIN is that PIN requires symbols for its analysis. This is the leading reason why CACEE opts to avoid statically linked binaries and other such binaries that lack dynamic symbols. Without these symbols, CACEE cannot perform its rules analysis, as library functions cannot be recognized. In a perfect world, CACEE could recognize functions taking their checksum and comparing those to a large database, but this exhaustive effort only handles a small portion of malware samples and is beyond the scope of this work.

An alternative debugger to PIN could be WinDbg. WinDbg, paired alongside hexray's IDA Pro disassembler, offers an extremely powerful interface that for binary analysis [10, 37]. In our own work, we found the overhead associated with these two tools was too cumbersome for large-scale analysis, and thus resorted to using PIN+Triton instead.

4.2.3 Triton

Triton is another dynamic binary analysis framework that is built from Intel's PIN, but provides numerous quality of life features, most notably bindings to Python2. In addition to a general purpose dynamic binary instrumentation platform, Triton also functions as a taint-analysis engine and is a symbolic execution engine. Unlike other symbolic execution engines, Triton operates on the CPU instructions themselves and does not implement its own search strategies [38, 36]. Triton's more lightweight approach to concolic analysis is more favorable to CACEE whose pipeline is performance sensitive. Furthermore, Triton implements its own tracer, which is used in CACEE to construct the program trace, which contains the data dependence of each CPU instruction executed.

There are a few issues with Triton, the foremost of which is that Triton does not support every x86 CPU instruction. Some complex CPU instructions must be ignored by the tracer leaving gaps within the program trace. Such gaps are typically minor, and do not cause issues with abstracting behaviors from the program trace. It is possible, but extremely tedious, to modify the Triton source to handle more x86 instructions, but such instruction modeling does not typically increase the scope of the system.

Another issue with Triton is the naivety of its symbolic execution platform. Complex buffer operations and pointer arithmetic require additional support. Furthermore, Triton does not offer built-in support function models like other engines. Despite these shortcomings, its lightweight approach is still ideal for CACEE.

Other alternatives to Triton are Angr and KLEE [16, 39]. Angr is the most widely used concolic execution platform with bindings in Python, while KLEE leverages LLVM compiler infrastructure for its symbolic execution engine. These two tools have been around for awhile, but each have their own issues that led to Triton becoming the best fit.

4.2.4 Python

Python is the preferred programming language for CACEE. This is because of its forgiving syntax, and importantly its ability to create bindings in C. The compiled bindings greatly increase system performance since the python interpreter is not required at runtime. Python 2 is the version used, since that is the only version supported by Triton at the moment. CACEE implements its own fork of the python programming language to compile necessary packages, notably networkx [40].

4.2.5 Z3

Z3 is an SMT (Satisfiability Modulo Theories) solver engine from Microsoft Research [41]. It is the tool of choice because of its strong documentation and Python API. CACEE starts Z3 as a side process and feesd symbolic data to Z3 and back by localhost socket communication. CACEE is responsible for making queries to Z3, which serves as a backend for CACEE.

4.2.6 Other Tools

While not essential to the system pipeline, IDA Pro alongside WinDbg is frequently used to manually reverse engineer individual malware samples [37, 10].

4.3 Expanding the Triton Dynamic Analysis Framework

CACEE interacts with Triton to manage memory with the malware's own memory space. Triton also binds functionality between PIN and Python. Some additional bindings are needed to be exposed as well as some additional functionality needs to be added which requires modifying the Triton source code. In addition, support for symbolic pointers and dynamically loaded symbols are needed for CACEE to fully explore all paths within a binary.

4.3.1 Exposing Additional Bindings to Python

Python C API (not to be confused with CPython, the whole language compiled into C) offers ways to converting complex dynamically typed python variables into C variables through **PyObjects**. Any Python data type passed to Triton from CACEE is treated as a **PyObject**. The function is then responsible for interpreting the **PyObject** into a usable data type in C++.

A simple example of a new binding exposed is *startAnalysisFromSymbols*. This function takes in a list from Python of function names such as *InternetConnect* or *WriteFile* that when encountered cause CACEE to begin tracing. This is put into place to boost performance by limiting the size of the trace. The **PyObject** is first interpreted into a **const char *** array with *PyStr_AsString*. Next, the function checks to see if there is a dynamic symbol with the same name as the function name. Finally, the function calls *RTN_Open*, *RTN_InsertCall*, and *RTN_Close* from the PIN API to start the tracer [42].

4.3.2 Adding Support for Symbolic Pointers

Native Triton does not support symbolized pointers and always concretizes them. Operations involving pointer arithmetic frequently result in symbolic memory access; it becomes necessary to mitigate this issue by introducing I/O protections for symbolic pointers [29]. In practice, this is a callback from Triton's symbolic memory framework. In this callback, if-then-else statement chains on the symbolic memory region iteratively. To determine the memory region size, we estimate using either the program trace or the program stack, depending on if the pointer was allocated dynamically or statically respectively. Should these size estimations fail and the malware attempt to access symbolic memory, then the region is concretized [29].

Table 4.1: Rules for modeling behaviors within the cloud-abusing malware dataset.

Rule	Short Description
Service	The name of the web service contacted.
API key	An API key or other identifier sent to the web service.
Downloader	The malware downloads a file to disk.
Dropper	The malware downloads a file then executes it.
Exfiltrator	The malware uploads data to the web service.

4.3.3 Adding Support for Loaded Symbols

Many malware samples, especially compressed malware samples, will dynamically load libraries and functions. Two functions in the Windows C++ API handle this; *LoadLibrary* loads a shared library with a specified name as a new image and *GetProcAddress* finds the address in memory of a function with a specified name [42]. These functions are instrumented with PIN by default to load image (IMG) and routine (RTN) objects respectively as these functions appear. While Pin may be aware of the new symbols, triton must tell CACEE about these new symbols in order for CACEE to adapt to new critical functions being loaded in. When these functions are loaded, triton will update CACEE's import tables. The import table, a dictionary, is passed into to Triton as a *PyObject*. A new dictionary entry is added through *PyDict_SetItem*, then returned back to CACEE.

4.4 Creating Dataset Rules

Since the end goal of CACEE is to extract behaviors from cloud-abusing malware, custom rules must be put into place. Two types are rules are implemented, web rules, which focus on characteristics of the web service, and behavior rules, which focus on what the malware does before and after interacting with the web service. Only web rules are required to interact with public web services; behavior rules may interact with C&C servers or other domains. Rules vary in complexity, and are summarized in Table 4.1.

Table 4.1 represents the various behaviors CACEE can potentially extract from a mal-

ware's trace. For each rule, there are many possible API sequences that could be used to satisfy their structures. Since implementing cases for all possible API sequences is very exhaustive, preferential modeling was performed based on the frequency of dynamic symbols present within the dataset. Static scans of malware repositories showed that few samples had dynamic symbols for socket functions, while the libraries *Wininet.lib* and *WinHttp.lib* were common. As such, function models and rule structures for the aforementioned libraries were implemented instead of models for *Winsock2.lib*.

Furthermore, internet API sequences are split into Complex Internet Sequences (CISs) and Simple Internet Sequences (SISs) depending on the malware author's implementation. CISs require more symbolic solving and thus are less likely to resolve than an SIS. CISs also often require packet reconstruction. An example of an SIS is the function *URLDownloadToFile* which completes an HTTP GET request with only one API call. On the other hand, CISs require many more functions and are often non-determinate in length.

Lastly, all rule implementations specific to this dataset are inherited from the *Rule* abstract class. The rule API runs outside of the concolic analysis, and runs in Python 3 instead of Python 2. The rules API only requires a program trace (saved in JSON format) as an input. Note that since the rules are implemented in post-analysis, they can be swapped out without having to rerun the samples.

4.4.1 Service Rule

The service rule is a web rule only intends to determine which web service the malware is attempting to contact; more precisely, it is looking for public web domains.

Service Structure

The service rule structure consists of any function that can be used to start a TCP session to a web domain or IP address. The most common functions used here are *InternetConnect*, *WinHttpConnect*, and *URLDownloadToFile*. Since there is no other information needed

from the function parameters other than a domain, each structure for the service rule is a single node from either a SIS or a CIS.

Service Method

As for a method, the root domain is first extracted with regular expressions. Then, the root domain is compared against Alexa's Top 10000 sites [26]. If the site is in the Top 10000, it is considered a public web service, and the rule is satisfied.

4.4.2 API Key Rule

The API Key rule is web rule that is much more complex than the service rule. Its purpose is to associate the abused web service with some sort of API key, user account, or uniform resource identifier. Unfortunately, authentication and authorization varies widely from service to service, and are often implemented in CISs. CACEE currently handles identifying information in additional HTTP headers or from within the domain itself.

API Key Structure

The structure for the API Key rule is often a CIS. A malware can make any number of calls to *WinHttpAddRequestHeaders* or similar functions and still maintain a valid request. On the other hand, the identifier could be within the domain itself, such as a link to a file sharing site or social media account. In the latter case, the structure is identical to that of the service rule. In the more complex former case, a larger section of the API sequence is required such as shown in Figure 4.1. This is because in general cases, CACEE opts to reconstruct the packet from the API sequence.

API Key Method

The method function of the API key rule performs the parsing of the reconstructed packet. It then uses regular expressions to extract any identifiers that it can find. This rule will

```c
#include <Windows.h>
#include <WinInet.h>
#pragma comment(lib, "wininet.lib")

int main()
{
    LPCWSTR user_agent = L"Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2)";
    HINTERNET hOpen = InternetOpen(user_agent, INTERNET_OPEN_TYPE_DIRECT, NULL, NULL, 0);
    HINTERNET hConn = InternetConnect(hOpen, L"www.gatech.edu",
        INTERNET_DEFAULT_HTTP_PORT, NULL, NULL, INTERNET_SERVICE_HTTP, 0, 0);
    HINTERNET hHttpOpen = HttpOpenRequest(hConn, L"GET", L"/", NULL, NULL, NULL, 0, 0);
    // HttpAddRequestHeaders(httpopen_handle, ...)
    HttpSendRequest(hHttpOpen, NULL, 0, NULL, 0);
    return 0;
}
```

(a) A code snippet that shows the generic sequence of an HTTP GET request to http://gatech.edu.

```
GET / HTTP/1.1
User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2)
Host: www.gatech.edu
Cookie: AWSALB=E/g1tPR4lPb7Da9YpZHMbAOYpOuBSO8qNrdgZp3ppKgXjtbW4SZ/GOhnXCIygsy3qG/JfMm9XGXxXDb1UqwISYkqZIdn6NDLoSBpWsGNCui5WnWye1DKJ++34Ll0; AWSALBCORS=E/
g1tPR4lPb7Da9YpZHMbAOYpOuBSO8qNrdgZp3ppKgXjtbW4SZ/GOhnXCIygsy3qG/JfMm9XGXxXDb1UqwISYkqZIdn6NDLoSBpWsGNCui5WnWye1DKJ++34Ll0

HTTP/1.1 301 Moved Permanently
Server: CloudFront
Date: Tue, 20 Apr 2021 18:04:32 GMT
Content-Type: text/html
Content-Length: 183
Connection: keep-alive
Location: https://www.gatech.edu/
X-Cache: Redirect from cloudfront
Via: 1.1 c7fba59f51c522169e5b1f2507f356ea.cloudfront.net (CloudFront)
X-Amz-Cf-Pop: ATL56-C1
X-Amz-Cf-Id: aJYZhK14JyXNZEl4T2Z6xJu3GWSEXmBs-a2rtOrrg1IN5N0U3rvBWQ==

<html>
<head><title>301 Moved Permanently</title></head>
<body bgcolor="white">
<center><h1>301 Moved Permanently</h1></center>
<hr><center>CloudFront</center>
</body>
</html>
```

(b) The resulting HTTP GET stream from the code sequence. There is also a DNS stream over UDP not shown.

Figure 4.1: The code for a HTTP GET request using WinINet and the resulting packet.

also examine the target host and any additional headers. In addition, the API key will track HTTP verbs and endpoints to categorize API abuse.

4.4.3 Downloader Rule

The downloader rule is a behavior rule, like the API key rule, it parses non-deterministic blocks of internet API sequences. More specifically, the downloader rule involves data dependence between data received from an HTTP GET or FTP sequence and writes to disk.

Downloader Structure

The downloader structure mirrors the API key structure at first. A multitude of structures including the sequence shown in Figure 4.1a can potentially represent the downloader rule structure. *URLDownloadToFile* is a simpler case that is a single node structure. CISs for the downloader structure tend to include the functions *InternetReadFile* or *WinHttpReadData* with data dependence to file API functions. Notably, it is very common occurrence to see some kind of string processing on the buffers written to by the aforementioned functions. These operations do not manifest themselves within the downloader structure but are considered in the rule method.

Downloader method

The downloader rule is almost entirely structure based. The only checks are to make sure that the data written to disk is originally sourced from the internet, that the buffer is not empty, and that the disk write operation succeeded.

4.4.4 Dropper Rule

The dropper rule is a behavioral rule that is an extension of the downloader rule— the malware downloads a file and then executes it. This behavior mirrors that of malicious

installers, often referred to as droppers.

Dropper Structure

Like the downloader rule, the dropper rule tends to start with a CIS then moves onto file API functions. As an extension, the dropper rule will also execute the downloaded file. No single node structures for this rule exist. Similar to the downloader rule, the structure is minimal; that is, it includes in order: the internet sequence, the file I/O command (if any), and then the process creation command.

Dropper Method

The dropper method requires several checks to make sure the behavior is correctly flagged. First, the file downloaded must be data dependent on the file API command. This is more difficult in practice; this is because many executable files are too large to fit into memory and must be downloaded to disk in fragments. While the internet sequence and file I/O are connected through data dependence; it is often the case where the process creation commands shares no data dependence with the previous API calls within the rule. To compensate for this, CACEE compares filenames used by the process creation commands and compares those to the filenames present within the file handles from the file I/O API call. If the filenames match (i.e. the same file is executed as is downloaded) then the rule is satisfied.

4.4.5 Exfiltrator Rule

The opposite of the downloader rule, the exfiltrator rule indicates that the malware is uploading data from the infected machine over the internet. It is a behavior rule that first requires a read from the hard drive then starting either a CIS or a SIS. If the file was not present at the start of execution, the exfiltration rule is still flagged.

Exfiltrator Structure

The structure of the exfiltrator rule involves two nodes. The first is a read operation with a file I/O API call, such as *fread* or *ReadFile*. The second sequence is an internet sequence that sends the read-in data over the network, most commonly an HTTP POST packet. Any string processing or other modifications made to the data before it is sent out is not considered within the structure.

Exfiltrator Method

The method of the exfiltrator rule is not very complex. In short, the method checks to see if there is data dependence from the file I/O API call and the network operation. If this is true, and the HTTP verb is POST or PUT (or FTP equivalent), then the rule is satisfied.

CHAPTER 5

EVALUATION

The small scale evaluation considers three malware samples: Vidar. a spyware that abuses ip-api.com, LOWBALL, a dropper malware abusing dropbox.com, and Vflooder, a malware trojan that reads from twitter to manipulate infected machines. Each sample is manually reverse engineered, which functions as the ground truth. Then, CACEE is run against each sample and their results are compared.

5.1 Vidar Case Study

Vidar (MD5: 40f19de327aac4b5d11d0882a7e67e9b) is a malware fork of the Arkei malware family [43]. It is a commercial malware that one can buy on shops and forums ranging from $250-700. The malicious C&C server is changed every version of Vidar, with this hash belonging to an unidentified early version. The malware is built on the marketplace server where it is configured by the buyer. A set of global flags exist which determine which functionalities are enabled or disabled. Once the malware has been configured, it is compiled with a user ID number and ready to deploy. The buyer can check the state of their infected machines on the rotating domain.

5.1.1 Vidar Summary

The Vidar malware, written in C++, starts by checking the user locale. If the locale is Russian, Azerbaijani, Belarusian, Kazakh, or Uzbeki, then the malware will not infect the machine. The malware generates a creates a hidden, temporary directory with a randomized name in the `C:\ProgramData\` parent directory. The first internet connection the malware makes is an HTTP connection to a Russian C&C server **immortalled.mcdir.ru** (alive as of publication). The malware generates a POST request using part of the machine

GUID as the identifier for the infected host. Vidar then moves on to download the specific libraries it needs to execute the rest of its payload. The malware then proceeds to collects a wide variety of information about the infected machine, including hardware information, running processes, and browser data. Next, Vidar obtains network information by sending an HTTP POST request to **ip-api.com**. Vidar uses this site to get JSON data about the host's network and dumps that information into a file called **information.txt**. The malware finalizes it search looking for cryptocurrency wallets as well as some user and system information. After this is done, Vidar compresses all of the data it finds and sends it back to its C&C server **immortalled.mcdir.ru**. Vidar presents very few anti-RE techniques, only really implementing some lightweight string obfuscation.

Note, more payloads are equipped in this malware, they are just disabled by configuration. In addition, the malware contains an updating dropper agent since the market vendors rotate the domain every version. This agent is also capable of injecting custom payloads from the buyer.

5.1.2 Web Profile for IP API

It is interesting to note that Vidar is using ip-api.com in the intended way. It is gathering information about the victim's IP address without using any kind of authorization or authentication. In addition to using no token, ip-api.com does not support uploading any text or media, and does not host user content. While one could consider the usage benign, any API use done by malware is considered malicious. Cases like the Vidar family and ip-api.com represent an emerging pattern in spyware.

5.1.3 Analysis on Vidar

CACEE's analysis of Vidar starts from the symbol *InternetOpenA* (A for ANSI encoding). This is the first API call in Vidar's first HTTP POST packet sent to **immortalled.mcdir.ru**. CACEE misses Vidar checking system locales and creating its own directory, neither of

```c
#include <Windows.h>
#include <WinInet.h>
#pragma comment(lib, "wininet.lib")

struct serverInfo {
    LPCSTR user_agent;
    LPCSTR serverName;
    LPCSTR objectName;
    LPCSTR username;
    LPCSTR password;
};

void myfunc(serverInfo* inputClass)
{
    // This code is adapted from the hex-rays decompiler output
    HINTERNET hOpen = InternetOpenA(inputClass->user_agent, INTERNET_OPEN_TYPE_PRECONFIG, NULL, NULL, 0);
    if (hOpen)
    {
        InternetSetOptionA(hOpen, INTERNET_OPTION_HTTP_DECODING, (LPVOID)1, 4); // Enables decoding gzip data
        HINTERNET hConnect = InternetConnectA(hOpen, inputClass->serverName,
            INTERNET_DEFAULT_HTTP_PORT, inputClass->username, inputClass->password, 3, NULL, 1);
        if (hConnect)
        {
            InternetSetOptionA(hConnect, INTERNET_OPTION_HTTP_DECODING, (LPVOID)1, 0); // Disables decoding gzip data
            HINTERNET hHttpOpen = HttpOpenRequestA(hConnect, "POST", inputClass->objectName,
                NULL, NULL, NULL, INTERNET_FLAG_KEEP_CONNECTION, 1);
            // function continues here
        }
    }
}
```

Figure 5.1: Decompiled code from the Vidar malware simplified for readability.

which have are important to the five rules in use for this sample. Analysis continues and CACEE finds a call to *InternetConnectA* as well as *HttpOpenRequestA*. A section of simplified decompiled code for Vidar is shown in Figure 5.1; CACEE is able to extract the parameters passed to these internet calls and store them in the program trace. Notably, CACEE is able to capture the domain for this version and the user profile ID (11), which indicates the buyer on the C&C server. Due to environmental constraints, *HttpSendRequestA* fails despite the C&C server being online. Despite not having access to its libaries, Vidar continues executing. CACEE executes pass the HTTP POST request to **ip-api.com** and to the final POST request to the C&C server. After this is finished, CACEE rewinds the state in an attempt to reach the *ShellExecuteA* API call which could not be reached concretely. This is the case where the malware authors have pushed an update or that the buyer has dropped a custom payload. CACEE instantiates a symbolic buffer for the return of *InternetReadFile* and execution continues to *CreateFileA* and *WriteFile*. Unfortunately, CACEE does not recognize a solvable branch from this point onward and closes.

In post-analysis, CACEE flags three rules satisfied: the service rule, the downloader

rule, and the exfiltrator rule. The service rule is satisfied first with a call to *InternetConnectA* passing the domain **ipapi.com**. Next, the exfiltrator rule is satisfied when Vidar calls *ReadFile* to read-in a gzip compressed **information.txt** file. The buffer from *ReadFile* has data dependence with *HttpAddRequestHeadersA*, which then shares handles with *HttpSendRequestA*. Finally, the downloader rule is satisfied since at the end of execution, CACEE is able to solve a branch from *InternetReadFile* eventually leading to *WriteFile* for the downloading behavior of Vidar. While Vidar exhibits the dropper behavior as well, it is not recognized by CACEE. The profile ID configured with this Vidar sample (11) is used as an additional header, but is not considered an API key since it is not used for a public service.

5.2 LOWBALL Case Study

LOWBALL (MD5: d76261ba3b624933a6ebb5dd73758db4) is a dropper malware used as a payload for a spear-phishing campaign targeting media companies in Hong Kong in 2015 [44]. Only 4% of the size of Vidar, LOWBALL is a backdoor trojan utilizing dropbox.com as its C&C server. The no longer existing Dropbox account was managed by the malware authors outside of this sample.

5.2.1 LOWBALL Summary

The first step in LOWBALL's execution is to deobfuscate its own hardcoded Dropbox API key. The malware then dynamically loads in *shell32.dll* and from it grabs the function *ShellExecuteA*. After this, the malware attempts to download an updated version of itself using an outdated version of the Dropbox API (**api.dropbox.com**). Should this download succeed, the malware would write the file to disk, then execute it as a new process. If the download fails, it enters a while loop waiting for the Dropbox account to come back online. In a new process, LOWBALL will send an HTTP PUT request creating a new file in the Dropbox folder with the hostname of the infected machine. Finally, the malware enters

an infinite loop where it communicates with the Dropbox account functioning as a C&C server. Interestingly, LOWBALL does not create a reverse shell of any kind; rather, it only uses batch files to control the infected machine. It is suspected that the malware authors hosted other malware in their Dropbox folder, with LOWBALL acting as a first stage [44].

5.2.2 Web Profile for Dropbox

Dropbox's flexible service can be very useful for malware authors wishing to obfuscate their network traffic. Its authentication scheme is by an API key passed an additional HTTP header. Dropbox allows the user to host almost any type of file; although it does have mitigations to prevent users from uploading malware [45]. Non-executable files such as shell commands can also be uploaded by the malware authors and used to control infected hosts. Dropbox does also offer a public front-end, since folders can be made public for anyone to download. All things considered, the Dropbox API can provide enormous utility to malware authors.

5.2.3 Analysis on LOWBALL

CACEE begins its analysis on the LOWBALL malware in its update function with a call to *InternetConnectA* used with domain **api-content.dropbox.com**. CACEE is able to follow the stream and the malware is successfully able to download an error message from the Dropbox account. LOWBALL uses *strncmp* to see if it received an error of any kind. It then proceeds to enter the while loop. CACEE detects that it cannot concretely exit the while loop, and rewinds the state to a solvable branch from the result of *strncmp*. It solves the branch then writes the error code using to disk using *fwrite*. The malware then proceeds to attempt to execute the downloaded file with *ShellExecuteA*. After this point LOWBALL would exit, but CACEE rewinds to a previous state and begins executing towards the main payload of the LOWBALL malware. CACEE catches several GET, PUT, and POST requests to the Dropbox API, but none resolve. LOWBALL does read in a file

called `onlineflagfile` and sends it by a POST request. CACEE eventually times out in the backdoor infinite loop, since there is no escaping branch.

In the post-analysis phase, CACEE flags all five rules as met for LOWBALL. The service rule is satisfied by several HTTP connections to **api-content.dropbox.com**, and the API Key rule is satisfied by **sgKddaX_ntAAAAAAAAAADVYeex9Pc0NuhGII10uLUhy-Kte7gEehQSxjYgRB2yWT**. CACEE sees that executes a downloaded file with *InternetReadFile*, *fwrite*, then *ShellExecuteA*, which satisfies both the dropper rule and the downloader rule. Lastly, CACEE identifies data dependence between a call to *fread* and a call to *HttpSendRequestA*, satisfying the exfiltrator rule.

5.3 Vflooder Case Study

Vflooder (MD5: 6b32a7ad0c62a86dc8d08f807d20b2a9) is a malware family that is designed to disrupt services by spam flooding APIs [46]. The attacked APIs vary family to family; in this case the victim APIs are VirusTotal and Twitter.

5.3.1 Vflooder Summary

The malware begins by creating a new thread using *CreateThread* passing the *main* function. After this the malware takes its own file into memory from disk. The malware then makes an API request to **virustotal.com** request via HTTP POST to **/vtapi/v2/file/scan** to submit a new file. The purpose of this is to perform a Denial-of-Service attack on VirusTotal by submitting copies of itself in a loop. Alongside this thread, Vflooder connects to a hardcoded twitter account (**/pidoras6**) and make a GET request via HTTPS. The malware will then parse the HTML from the twitter page to extract a tweet from the malware author, decode the tweet from base64 and jump to its own C&C server **https://w0rm.in/join/join.php**, which is no longer active. According to MalwareBytes, this site once sold website exploits [46]. The malware then proceeds to read data in from the malicious website with *WinHttpReadData*.

5.3.2 Web Profile for VirusTotal

While at first one might think that it is best for malware to avoid VirusTotal at all costs, it is still possible to abuse VirusTotal's API. For authentication and authorization, VirusTotal uses an API key passed as an additional header that carries privileges with it. For file sharing, it is possible to upload files to VirusTotal for scans, and some files may be downloaded depending on permissions. Text sharing is also possible through comments. Lastly, VirusTotal does have a few ways to host user data (e.g. a user could manage comments on a filehash) but users are extremely limited in ways they can manage it [25].

5.3.3 Web Profile for Twitter

Twitter's API has a history of abuse from a multitude of misinformation campaigns [47]. Like VirusTotal, it requires an API key passed as an additional HTTP header, which has permissions tied to it. Twitter allows primitive file sharing in the form of tweets, including images, but not executables [32]. Text sharing is very easy on twitter in the form of tweets. Perhaps most of all, twitter has a powerful public front-end in the form of malicious accounts.

5.3.4 Analysis of Vflooder

CACEE begins analysis of the Vflooder malware from the *CreateThread* function. It tracks the DoS thread's HTTP communications with VirusTotal, and extracts the malware's hard-coded API-key. Analysis then proceeds to the jump-server portion of the malware. CACEE identifies a GET request to twitter.com which resolves but is blocked on the virtual machine network. Vflooder, with incorrectly parsed HTML, makes an API call to *WinHttpCrackUrl* which is then interpreted as a search engine request for the less-than-character <. At this point, the malware crashes and CACEE cannot continue analysis.

Despite a flawed execution, CACEE still maintains a context-rich execution trace for this strain of Vflooder. The service rule is satisfied twice, once by VirusTotal and again by

Table 5.1: CACEE analysis for web rules on three case studies.

Malware Sample	Service	API Key
Vidar	ip-api.com	None
LOWBALL	api-content.dropbox.com	sgKddaX_ntAAAAAAAAAADV...
Vflooder	twitter.com virustotal.com	None a0283a2c3d55728300d06487...

Table 5.2: CACEE analysis for behavior rules on three case studies.

Malware Sample	Downloader	Dropper	Exfiltrator
Vidar	GET /line	None [1]	POST US_636d1cd5...
LOWBALL	GET /1/files/auto	WmiApCom.bat	POST onlineflagfile
Vflooder	GET /pidoras6	None	POST 0933a85ab... [2]

Twitter. The API key rule is also satisfied, since the VirusTotal API key is extracted as an additional header passed to *WinHttpSendRequest*. The downloader rule is also satisfied by a GET request to the twitter account. The exfiltrator rule is mistakenly marked as satisfied, as Vflooder reads itself from disk with *ReadFile*, and posts to VirusTotal, and not machine specific data as with the case of the other two samples.

5.4 Findings

In Table 5.1, CACEE is able to correctly identify every web characteristic of the public web service according to the rules provided. This includes each public web service that the malware connected to, and any credentials the malware used that were associated with that API. For the behavior rules, shown in Table 5.2, CACEE misses that the Vidar sample can download and execute files, and falsely identifies that the Vflooder malware is exfiltrating data from the infected machine. The rules reflect behaviors that are expected to be observed in the dataset with manual reverse engineering. It is possible in a large scale study of samples with CACEE could analyze malware to a similar effect.

[1] False Negative
[2] False Positive

CHAPTER 6
CONCLUSION

6.1 Related Works

Concolic execution has already seen applications for malware analysis for windows malware. Brumley et al. used concolic execution to identify trigger conditions in malware, including missing C&C servers, a condition that has been frequently observed within our dataset [48]. Other frameworks like S2E aim to extract a trace from the sample with concolic execution, which CACEE more closely resembles [17]. S2E even creates a dedicated analysis environment based on preset Windows images. Pure symbolic engines like Angr read in a memory dump and solve for constraints without the use of concrete execution [16]. Older research, such as Qi et al., uses concolic execution for code coverage and pathfinding [14].

With specific regard to Remote Access Trojan (RAT) malware, Baldoni et al. used concolic execution to emulate C&C server commands [49]. Furthermore, Moser et al. were even able to recreate C&C servers with concolic execution on RAT malware [50]. Shankarapani et al. used API call patterns taken from static analysis to create fast signatures for malware identification [51]. On the other hand CACEE does not aim to identify malware with signature based detection, but rather analyze known malware samples to gain a greater understanding of the malware's payload. CACEE also differs from existing concolic execution tools in that it attempts to automate much of the data modeling process.

6.2 Limitations

The development of CACEE is still in its infancy, and as such there are many corner cases that CACEE cannot yet handle. As previously discussed in Table 3.1, there are many anti-

RE techniques greatly limiting the capabilities of CACEE on sophisticated malware samples. The prevalence of these techniques and packing tools that implement them reduce the generality of CACEE. Beyond this, CACEE is currently only implemented for Windows-PE32 malware on the x86 architecture, and does not support the .NET framework. Other architectures or operating systems are not currently supported at this time, further limiting generality.

In addition to dataset limitations, CACEE relies on API calls for branch conditions. That is, not all branches are solvable for CACEE, as it relies on function modeling to introduce symbolic data. Branch conditions that exist before the first encountered critical symbol are also not subject for forward solving. Finally, not all complex data types are modeled yet, or are even exposed in the Windows header files. As a result some branch conditions dependent on these complex data types cannot be solved.

Unlike other concolic execution engines, while tolerant of the path explosion problem, CACEE struggles to interpret data types outside of API calls. This is because CACEE solves branch conditions from the EFLAGS register and introduces symbolic memory regions as a result. To do this, CACEE must know the size of the return value of the API call. This is a difficult problem at the binary level with complex memory objects of variable size, since complex memory structures often contain metadata indicative of their size.

6.3 Conclusion

The lifespan of malware in the wild is typically very short. To compensate for this problem, CACEE uses concolic execution to trick the malware into behaving as if it were released just a few days ago. Monitoring the data defined and used by the malware as it runs, CACEE dumps more information than a traditional sandbox albeit in a longer time. With this information in hand, CACEE abstracts behaviors from the data in order to provide high level summaries of the malware's payload.

Another tool in a malware analyst's toolbox, CACEE is specifically designed to tackle

an emerging trend in malware, the use of public web services in malicious payloads. Capable of handling a variety of anti-RE techniques, CACEE unloads vast amounts of information about a malware sample in an automated fashion. With this data, CACEE is able to understand trends in large datasets of malware and malware families. It is also possible to adapt CACEE behavior analysis to model behaviors expectant of different types of malware, such as internet-of-things malware or industrial control system malware.

In a few cases, CACEE has shown results comparable to that of manual reverse engineering. It has also shown itself capable of exploring paths not normally reachable through only concrete analysis. These results may indicate that in order to tackle emerging trends in malware a in-depth approach such as binary analysis may be required. Future trends, such as DNS-over-HTTPS, may further complicate network-oriented analysis and require binary analysis solutions such as CACEE and other concolic execution tools.

REFERENCES

[1] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel, "A view on current malware behaviors.," in *Proceedings of the 2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, Boston, MA, Apr. 2009.

[2] P. A. Networks, "Stop attackers from using dns against you," Tech. Rep., 2020.

[3] K. Ingham and S. Forrest, "A history and survey of network firewalls," 2014.

[4] M. Antonakakis, R. Perdisci, Y. Nadji, N. Vasiloglou, S. Abu-Nimeh, W. Lee, and D. Dagon, "From throw-away traffic to bots: Detecting the rise of dga-based malware," in *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, Aug. 2012.

[5] *Automating malware scanning for documents uploaded to cloud storage*, https://cloud.google.com/architecture/automating-malware-scanning-for-documents-uploaded-to-cloud-storage, [Accessed: 2021-03-28].

[6] Z. Li, S. Alrwais, Y. Xie, F. Yu, and X. Wang, "Finding the linchpins of the dark web: A study on topologically dedicated hosts on malicious web infrastructures," in *Proceedings of the 34th Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2013.

[7] *Evasions in intrusion prevention/detection systems*, https://www.virusbulletin.com/virusbulletin/2010/04/evasions-intrusion-prevention-detection-systems, [Accessed: 2021-03-28].

[8] I. Corporation, *Intel® 64 and ia-32 architectures software developer's manual*, 2021, p. 18.

[9] *User mode and kernel mode*, https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/user-mode-and-kernel-mode, [Accessed: 2021-03-29].

[10] D. Vostokov, *WinDbg: A Reference Poster and Learning Cards*. Opentask, 2008, ISBN: 190671729X.

[11] S. Vömel and F. C. Freiling, "A survey of main memory acquisition and analysis techniques for the windows operating system," *Digital Investigation*, vol. 8, pp. 3–22, 2011.

[12] A. Ferrari, *X86 assembly guide*, https://www.cs.virginia.edu/~evans/cs216/guides/x86.html, [Accessed: 2021-04-02].

[13] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," vol. 13, no. 4, pp. 451–490, Oct. 1991.

[14] D. Qi, H. D. Nguyen, and A. Roychoudhury, "Path exploration based on symbolic output," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 4, p. 32, 2013.

[15] Y. Li, Z. Su, L. Wang, and X. Li, "Steering symbolic execution to less traveled paths," in *Proceedings of the 2013 Annual ACM SIGPLAN International Conference on Object Oriented Programming, Systems, Languages & Applications (OOPSLA)*, Indianapolis, IN, Oct. 2013.

[16] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *Proceedings of the 37th Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[17] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," *ACM SigPlan Notices*, vol. 46, no. 3, pp. 265–278, 2011.

[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," *ACM SigPlan Notices*, vol. 10, no. 6, pp. 234–245, 2005.

[19] R. Felding, *Architectural styles and the design of network-based software architectures*, 2000.

[20] E. D. Hardt, "The oauth 2.0 authorization framework," RFC Editor, RFC 6749, Oct. 2012.

[21] *Http request methods*, https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods, [Accessed: 2021-04-27].

[22] *Open movie database*, https://www.omdbapi.com/, [Accessed: 2021-04-28].

[23] D. Connolly and L. Masinter, "The 'text/html' media type," IETF, RFC 2584, Jun. 2000.

[24] G. J. Széles and A. Coleşa, "Malware clustering based on called api during runtime," in *Proceedings of the International Workshop on Information and Operational Technology and Security (IOSec)*, Crete, GR, Sep. 2018.

[25] *Virustotal corporation*, https://www.virustotal.com/, [Accessed: 2021-04-02].

[26] *The top 500 sites on the web*, https://www.alexa.com/topsites, [Accessed: 2021-04-02].

[27] *Windows 32 api reference*, https://docs.microsoft.com/en-us/windows/win32/apiindex/windows-api-list, [Accessed: 2021-04-15].

[28] S. Mittal, *A survey of techniques for dynamic branch prediction*, Apr. 2018.

[29] F. Kilger, "Extracting ics models from malware via concolic analysis," M.S. thesis, Georgia Institute of Technology, 2020.

[30] *Understanding sal*, https://docs.microsoft.com/en-us/cpp/code-quality/understanding-sal?view=msvc-160, [Accessed: 2021-04-06].

[31] *Ip geolocation api*, https://ip-api.com, note=[Accessed: 2021-04-15.

[32] *Twitter api*, https://developer.twitter.com/en/docs/twitter-api, [Accessed: 2021-04-21].

[33] *Dropbox inc.* https://dropbox.com, [Accessed: 2021-04-15.

[34] *Qemu: Quick emulator*, https://github.com/qemu/QEMU, [Accessed: 2021-04-19].

[35] *Virusshare*, https://virusshare.com/, [Accessed: 2021-04-20].

[36] F. Saudel and J. Salwan, "Triton: A dynamic symbolic execution framework," in *Proceedings of the Information and Communications Technology Security Symposium (SSTIC)*, Rennes, France, 2015, pp. 31–54.

[37] *Ida pro - interactive disassembler*, https://www.hex-rays.com/IDA-pro/, [Accessed: 2021-04-19].

[38] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," *ACM SigPlan Notices*, vol. 47, no. 6, pp. 193–204, 2012.

[39] C. Cadar, D. Dunbar, D. R. Engler, *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.

[40] *Networkx: Network analysis in python*, https://networkx.org/, [Accessed: 2021-04-19].

[41] *Z3: The z3 theorem solver*, https://github.com/Z3Prover/z3, [Accsesed: 2021-04-19].

[42] *Pin 3.18 user guide*, https://software.intel.com/sites/landingpage/pintool/docs/98332/Pin/html/, [Accessed: 2021-04-19].

[43] *Let's dig into vidar – an arkei copycat/forked stealer (in-depth analysis)*, https://fumik0.com/2018/12/24/lets-dig-into-vidar-an-arkei-copycat-forked-stealer-in-depth-analysis/, [Accessed: 2021-04-20].

[44] *China-based cyber threat group uses dropbox for malware communications and targets hong kong media outlets*, https://www.fireeye.com/blog/threat-research/2015/11/china-based-threat.html, [Accessed: 2021-04-20].

[45] *How dropbox handles viruses and malicious software*, https://help.dropbox.com/accounts-billing/security/viruses-malware, [Accessed: 2021-04-20].

[46] *Analyzing malware by api calls*, https://blog.malwarebytes.com/threat-analysis/2017/10/analyzing-malware-by-api-calls/, [Accessed: 2021-04-21].

[47] S. Lee and J. Kim, "Warningbird: A near real-time detection system for suspicious urls in twitter stream," Feb. 2012.

[48] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin, "Automatically identifying trigger-based behavior in malware," in *Botnet Detection*, Springer, 2008, pp. 65–88.

[49] R. Baldoni, E. Coppa, D. C. D'Elia, and C. Demetrescu, "Assisting Malware Analysis with Symbolic Execution: A Case Study," in *Proceedings of the International Conference on Cyber Security Cryptography and Machine Learning (CSCML)*, Israel, Jun. 2017.

[50] A. Moser, C. Kruegel, and E. Kirda, "Exploring Multiple Execution Paths for Malware Analysis," in *Proceedings of the 28th Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2007.

[51] M. K. Shankarapani, S. Ramamoorthy, R. S. Movva, and S. Mukkamala, "Malware detection using assembly and api call sequences," *Journal in Computer Virology*, vol. 7, pp. 107–119, 2011.